UNIVERSIDAD NACIONAL AUTÓNOMA
DE MÉXICO

FACULTAD DE CIENCIAS

Lógica de Hoare para Programación Funcional

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

Licenciado en Ciencias de la Computación

PRESENTA:

Gilberto Isaac López García

TUTORA

Dra. Lourdes del Carmen González Huesca

Ciudad de México 2021

Tesis realizada bajo el proyecto PAPIME 102117
"Tópicos en Ciencia de la Computación Teórica"

Índice general

Índice de figuras **v**

Introducción **1**

1. Lógica de Hoare: el caso de la programación imperativa **3**

1.1. El lenguaje imperativo While 4

1.2. Evaluación, semántica operacional de paso grande 4

1.2.1. Semántica de *AExp* 5

1.2.2. Semántica de *BExp* 6

1.2.3. Semántica operacional de *Com* 7

1.3. Lógica de Hoare, semántica axiomática 9

2. Lógica de Hoare para un núcleo funcional simple **17**

2.1. El asistente de pruebas Coq 19

2.2. Gramática del lenguaje funcional simple 20

2.3. Semántica operacional de paso grande 20

2.3.1. Evaluación de programas P 22

2.3.2. Evaluación de expresiones aritméticas A 24

2.3.3. Evaluación de expresiones booleanas B 25

2.4. Lógica de Hoare 25

2.5. Implementación 35

3. El lenguaje PCF **41**

3.1. Representación local anónima 42

3.2. Definición del lenguaje PCF 43

3.3. Semántica operacional de paso grande para PCF 49
3.4. El sistema de tipos para PCF 53
3.5. Lógica de Hoare para PCF 56
3.5.1. Reflexión lógica de tipos y valores 57
3.5.2. Aserciones y Ternas de Hoare 59
3.5.3. El sistema de inferencia 61

Conclusiones **69**

Apéndice A. Representación local anónima **73**

Bibliografía **79**

Índice de figuras

1.1. El lenguaje WHILE (sintaxis) 5

1.2. La lógica de Hoare para el lenguaje WHILE 15

2.1. El lenguaje funcional simple (sintaxis) 21

2.2. Relación de evaluación para programas 23

2.3. Relación de evaluación para expresiones aritméticas 24

2.4. Relación de evaluación para expresiones booleanas 25

2.5. La lógica de Hoare para el lenguaje funcional simple 36

a. Reglas de inferencia para programas y funciones 36

b. Reglas de inferencia para constantes y variables 36

c. Reglas de inferencia para operadores binarios 37

d. Reglas de inferencia para operadores unarios 37

3.1. El lenguaje PCF (sintaxis) 43

3.2. Términos de PCF 45

3.3. Relación de evaluación para PCF 49

3.4. Los tipos de PCF 53

3.5. Juicios de tipificado para PCF 53

A.1. Cálculo lambda (sintaxis empleando la representación local anónima) 76

Introducción

La lógica de Hoare es un formalismo que nos permite llevar a cabo verificación de software: nos permite proporcionar especificaciones de programas y ofrece un método formal para demostrar la corrección de programas con respecto a sus especificaciones. Introducida hace poco mas de cincuenta años por el científico en computación Tony Hoare en su artículo *An axiomatic basis for computer programming* [11], y tomando inspiración de trabajos previos, notablemente el artículo de Robert W. Floyd *Assigning meanings to programs* [10], Hoare propuso un sistema de inferencia buscando capturar el razonamiento deductivo que se sigue al razonar con programas imperativos, argumentando que

> La programación es una ciencia exacta en el sentido que todas las propiedades de un programa y todas las consecuencias de su ejecución en un ambiente dado se pueden, en principio, averiguar a partir del texto del propio programa exclusivamente mediante razonamiento deductivo. El razonamiento deductivo implica la aplicación de reglas de inferencia válidas a conjuntos de axiomas válidos. Por lo tanto, es deseable e interesante dilucidar los axiomas y reglas de inferencia que subyacen nuestro razonamiento sobre programas de computadora. [...]

La lógica de Hoare ha sido objeto de estudio a lo largo de estos años, investigando extensiones al sistema para su aplicación en distintas áreas de la programación (concurrencia, no determinismo, orientación a objetos, etc. [1]) y sus propiedades teóricas utilizando lógica matemática. También ha disfrutado de gran popularidad como un herramienta práctica para la verificación formal de software en aplicaciones del mundo real, por ejemplo, el *Proyecto KeY* para verificación de programas escritos en Java, o el lenguaje Dafny. Sin embargo, el enfoque ha sido principalmente hacia la programación imperativa.

En el presente trabajo buscamos diseñar una lógica de Hoare para programación funcional tomando como inspiración los artículos *A Hoare logic for call-by-value functional programs* de Yann Régis-Gianas y François Pottier [18], y *A compositional logic for polymorphic higher-order functions* de Kohei Honda y Nobuko Yoshida [13]. Los autores de ambos trabajos notan que no se ha dedicado suficiente atención al estudio de la lógica de Hoare en el caso funcional a pesar de sus

ventajas sobre el paradigma imperativo, a saber: es más fácil razonar con programas funcionales por ser funciones en el sentido matemático, que razonar sobre estados, apuntadores/referencias, datos mutables y demás; y la posibilidad de proporcionar especificaciones de software y pruebas de corrección más simples.

La lógica de Hoare es un caso de semántica axiomática para lenguajes de programación. La semántica se ocupa de dar formalmente un significado a las construcciones sintácticamente correctas de un lenguaje y hay distintas formas de abordar esta tarea. Las formas más comunes en la literatura para hacerlo son:

- *Semántica Operacional*: El significado de un programa está dado por las operaciones o cómputos que induce en una máquina, esto es, el cómo se ejecuta. Aquí encontramos comúnmente dos estilos diferentes:
  - *De paso pequeño* que detalla la secuencia de pasos que ocurren en la máquina durante la ejecución, y
  - *De paso grande* que explica en general a qué resultados llega la máquina, dados el estado inicial y los resultados de las subexpresiones del programa (si estas existen)[1].
- *Semántica Denotativa*: El significado de un programa se da mediante objetos matemáticos (estos llamados denotaciones, ej.: funciones) que abstraen el efecto de su ejecución.
- *Semántica Axiomática*: Esta semántica hace algo distinto a lo que hacen las otras dos, aquí se proporcionan axiomas y reglas de inferencia que se asumen como la especificación formal del lenguaje, así el significado de un programa es toda aquella propiedad que se pueda demostrar con dichos axiomas y reglas.

Como herramienta para verificación formal, la lógica de Hoare es más poderosa y ofrece mayor seguridad que pruebas unitarias o análisis estáticos de código como el chequeo de tipos que realiza un compilador automáticamente. También es más flexible que asistentes de prueba como Coq [7] que ofrecen corrección total y permiten extraer programas que son correctos por construcción. Con la lógica de Hoare podemos describir rigurosamente, con mayor precisión que simples anotaciones de tipo, el comportamiento esperado de nuestros programas y demostrar formalmente que satisfacen sus especificaciones de manera composicional.

En el presente trabajo nos ocuparemos de presentar una semántica axiomática al estilo la lógica de Hoare, primero para un lenguaje funcional simplificado y después para el lenguaje PCF, junto con sus implementaciones [14] en el asistente de pruebas Coq.

[1] En la literatura, la semántica operacional de paso pequeño es llamada también *semántica estructural*, mientras que la de paso grande es llamada *semántica natural*. Es común encontrar una presentación de ambas especificaciones así como una demostración de su equivalencia.

Capítulo 1

Lógica de Hoare: el caso de la programación imperativa

La lógica de Hoare es un sistema formal para razonar sobre programas imperativos, originalmente presentado por C. A. R. Hoare en [11]. Esta lógica a veces también es llamada lógica de Floyd-Hoare por el trabajo de Robert W. Floyd en [10], cuyas ideas contribuirían en gran medida al trabajo de Hoare.

Los trabajos de Floyd y Hoare se ocuparon de proporcionar un significado a programas imperativos mediante un marco de trabajo con el cual poder escribir y demostrar propiedades de interés de estos programas, es decir, para dar una especificación y una demostración de la corrección de programas con respecto a sus especificaciones. Mientras que Floyd dio una serie de reglas para razonar sobre *flowcharts* (una gráfica dirigida cuyos vértices son los comandos de un programa y las aristas el flujo de las estructuras de control), las reglas que dio Hoare permiten razonar directamente sobre el texto de un programa, anotando comandos con predicados lógicos que describen relaciones entre (los valores de) las variables y que han de ser verdaderos antes y después de su ejecución.

Comenzaremos presentando el pequeño lenguaje WHILE, que cuenta con las construcciones sintácticas correspondientes a las estructuras de control comúnmente encontradas en el paradigma imperativo, definiremos la gramática y la ejecución de programas, entonces se presentará la lógica de Hoare para WHILE como lo hace Pierce en [17], un sistema deductivo que consiste de una serie de axiomas y reglas de inferencia que nos dicen cómo razonar sobre cada construcción sintáctica del lenguaje para poder demostrar de manera rigurosa que un programa posee cierta propiedad; de aquí el nombre *semántica axiomática*.

1.1. El lenguaje imperativo WHILE

Como ya mencionamos, WHILE es un lenguaje de programación imperativo. Para su definición requerimos las siguientes categorías sintácticas:

- *Num*, los números enteros,
- *Var*, nombres de variables,
- *AExp*, las expresiones aritméticas,
- *BExp*, las expresiones booleanas,
- *Com*, los comandos.

Asumimos que las definiciones de las categorías sintácticas *Num* y *Var* ya están dadas, por lo que no nos molestaremos en presentarlas aquí.

La gramática de WHILE está dada por las reglas en notación estilo BNF en la figura 1.1, donde n es una metavariable para un número, x para una variable, a para expresiones aritméticas, b para expresiones booleanas, C para comandos, *tt* y *ff* son las constantes booleanas *true* y *false* respectivamente.

1.2. Evaluación, semántica operacional de paso grande

En esta sección nos ocuparemos de darle un significado a los programas del lenguaje WHILE mediante una semántica operacional de paso grande. Más adelante se proporcionará una semántica axiomática para WHILE basándonos en lo que nos dice esta semántica operacional sobre lo que ocurre durante la ejecución de un programa.

La ejecución de los programas en WHILE dependen del valor de las variables que figuran en su texto, para ello definiremos lo que es un estado, que mantendrá los valores numéricos asociados a cada variable en un punto dado de la ejecución.

Un estado s es una función total de las variables en los números enteros

$$s : \mathit{Var} \to \mathbb{Z}$$

Dada una variable x en *Var*, y s un estado, $s(x)$ es el valor de x en s. Al conjunto o espacio de estados lo denotamos con S. El valor de una variable puede cambiar durante la ejecución de un programa, pero cuando una variable no haya sido asignada explícitamente se asumirá que su valor es 0.

			AExp
a	$::=$	n	*Número*
	$\mid$	x	*Variable*
	$\mid$	$a_1 + a_2$	*Suma*
	$\mid$	$a_1 * a_2$	*Producto*
	$\mid$	$a_1 - a_2$	*Resta*
			BExp
b	$::=$	tt	*true*
	$\mid$	ff	*false*
	$\mid$	$a_1 = a_2$	*Igualdad*
	$\mid$	$a_1 <= a_2$	*Menor o igual*
	$\mid$	$\sim b$	*Negación*
	$\mid$	$b_1 \,\&\, b_2$	*Conjunción*
			Com
C	$::=$	$x := a$	*Asignación*
	$\mid$	**skip**	*skip*
	$\mid$	$C_1; C_2$	*Secuencia*
	$\mid$	**if** b **then** C_1 **else** C_2	*Condicional*
	$\mid$	**while** b **do** C	*Ciclo*

Figura 1.1: El lenguaje WHILE (sintaxis)

1.2.1. Semántica de *AExp*

Dada una expresión aritmética a y un estado de variables s, definimos el significado de la expresión a en el estado s con la función total

$$A : \mathit{AExp} \times S \to \mathbb{Z}$$

definida recursivamente sobre la estructura de *AExp* como:

$$\begin{aligned} A(n, s) &= n \\ A(x, s) &= s(x) \\ A(a_1 + a_2, s) &= A(a_1, s) + A(a_2, s) \\ A(a_1 * a_2, s) &= A(a_1, s) * A(a_2, s) \\ A(a_1 - a_2, s) &= A(a_1, s) - A(a_2, s) \end{aligned}$$

Si $A(a, s) = n$ decimos que la expresión aritmética a en el estado s se evalúa[1] a n.

En la gramática de WHILE también figuran números enteros *Num*. Dado un número entero n, este es un término sintáctico de WHILE que asociaremos con un único elemento en $\mathbb{Z}$. Asumimos que la estructura e implementación de la clase sintáctica *Num* ya está dada y que esta asociación es una función bien definida. Con esto el primer caso en la definición de A tiene sentido.

Es claro que A es una función total, su definición depende del estado que por definición es total, y de las operaciones deterministas en $\mathbb{Z}$, por lo que dado un estado, una expresión aritmética siempre se evalúa a un número.

Nótese que en ningún punto de la definición de A se modificó el estado, esto es, el valor de las variables de un programa no cambia durante la evaluación de una expresión aritmética. Decimos entonces que las expresiones aritméticas no tienen efectos secundarios.

1.2.2. Semántica de *BExp*

Definimos el significado de una expresión booleana b en un estado s con la función total

$$B : \mathit{BExp} \times S \to \{\mathit{tt}, \mathit{ff}\}$$

definida recursivamente sobre la estructura de *BExp* como:

$$\begin{aligned}
B(\mathit{tt}, s) &= \mathit{tt} \\
B(\mathit{ff}, s) &= \mathit{ff} \\
B(a_1 = a_2, s) &= \begin{cases} \mathit{tt} & \text{si } A(a_1, s) = A(a_2, s) \\ \mathit{ff} & \text{e.o.c. (en otro caso)} \end{cases} \\
B(a_1 <= a_2, s) &= \begin{cases} \mathit{tt} & \text{si } A(a_1, s) \leq A(a_2, s) \\ \mathit{ff} & \text{e.o.c.} \end{cases} \\
B(\sim b, s) &= \sim B(b, s) \\
B(b_1 \ \& \ b_2, s) &= B(b_1, s) \ \& \ B(b_2, s)
\end{aligned}$$

Si $B(b, s) = \mathit{tt}$ decimos que la expresión booleana b en el estado s se evalúa a *tt* o *true*, o que la expresión es verdadera; si $B(b, s) = \mathit{ff}$ decimos que la expresión booleana b en el estado s se evalúa a *ff* o *false*, o que la expresión es falsa.

Análogamente a los números, supondremos que los valores booleanos *tt* y *ff* (como expresiones de WHILE) se corresponden con los valores de verdad *true* y *false* respectivamente y que se puede operar con ellos (negación y conjunción).

[1] Aquí escribimos "evaluación" para expresiones aritméticas y booleanas, para los comandos también escribiremos "ejecución".

Al igual que las expresiones aritméticas, las expresiones booleanas no tienen efectos secundarios y B es una función total. En ambos casos no nos interesa cómo se evalúan paso a paso estas expresiones, solo nos va a interesar el resultado de su evaluación.

1.2.3. Semántica operacional de *Com*

Una vez que hemos definido el significado de las expresiones aritméticas y booleanas que figuran en los comandos del lenguaje WHILE podemos definir el significado de los comandos.

Mientras que las expresiones en *AExp* y *BExp* se evalúan a un valor concreto dado un estado, los comandos modifican el estado, por lo que al ejecutar un programa en un estado inicial obtendremos (potencialmente) un estado final. Entonces la semántica de los comandos se dará mediante una relación entre el comando que se está ejecutando, el estado inicial y el estado final.

La relación para la semántica de comandos está dada por ternas (C, s, s') que escribimos

$$\langle C, s\rangle \longrightarrow s'$$

donde la *configuración* $\langle C, s\rangle$ representa el comando C que está por ser ejecutado en el estado s, y s' representa el estado resultante de dicha ejecución. Esto se lee "la ejecución de C en el estado s termina en el estado s'".

Es importante notar que no todas las configuraciones $\langle C, s\rangle$ llegan a un estado final s', esto es, no toda pareja de comando y estado termina su ejecución. Para ejemplificar esto consideremos el programa

while tt **do skip**

que intuitivamente es un ciclo infinito y no debería detenerse sin importar el estado s en que se ejecuta, por lo que nunca terminamos en un estado final s'. Esto será más claro pronto, cuando definamos la ejecución de comandos de la forma **while** b **do** C.

La definición de la relación $\longrightarrow$ está dada por una serie de reglas que se dan a continuación explicando qué ocurre en la ejecución de cada comando.

Al asignar el resultado de la expresión aritmética a en el estado s, a la variable x, el estado resultante s' se obtiene de tomar s y actualizar el valor de x por el resultado de la evaluación de a en s. Dado que a siempre se evalúa a un número, la ejecución de este comando siempre termina. Este nuevo estado s' lo escribiremos como $s[A(a, s)/x]$ y formalmente se define como

$$s[A(a,s)/x](y) = \begin{cases} A(a,s) & \text{si } y = x \\ s(y) & \text{e.o.c.} \end{cases}$$

entonces la evaluación del comando para asignación de variables está dada por

$$\frac{}{\langle x := a, s\rangle \longrightarrow s[A(a,s)/x]}\ \textsc{While-Eval-Asig}$$

El comando **skip** es un comando que no hace nada, su ejecución siempre termina y no modifica el estado, así que si el comando **skip** se ejecuta en un estado s, el estado resultante es s mismo. Así, la evaluación de **skip** está dada por

$$\frac{}{\langle \textbf{skip}, s\rangle \longrightarrow s}\ \textsc{While-Eval-Skip}$$

Para la secuencia de comandos $C_1; C_2$, se evalúa primero C_1, seguido se evalúa C_2. Si la ejecución de C_1 en un estado s termina en un estado s', en este estado s' se ejecuta C_2 y si termina en un estado s'', la evaluación de $C_1; C_2$ en s termina en s''

$$\frac{\langle C_1, s\rangle \longrightarrow s' \qquad \langle C_2, s'\rangle \longrightarrow s''}{\langle C_1; C_2, s\rangle \longrightarrow s''}\ \textsc{While-Eval-Sec}$$

En el caso de una expresión condicional **if** b **then** C_1 **else** C_2, en un estado s, primero se evalúa la guardia b en s, si esta es verdadera continúa solo la evaluación de C_1, en caso contrario solo la evaluación de C_2. Dado que las expresiones booleanas no tienen efectos secundarios el estado s no cambia y C_1 (o C_2 según sea el caso) se ejecuta en el estado s. Si termina en un estado s', este es el resultado de la evaluación de la expresión condicional. Debido a la guardia, tenemos dos reglas

$$\frac{B(b,s) = tt \qquad \langle C_1, s\rangle \longrightarrow s'}{\langle \textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2, s\rangle \longrightarrow s'}\ \textsc{While-Eval-Cond}_{tt}$$

$$\frac{B(b,s) = ff \qquad \langle C_2, s\rangle \longrightarrow s'}{\langle \textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2, s\rangle \longrightarrow s'}\ \textsc{While-Eval-Cond}_{ff}$$

La ejecución de un ciclo consiste en la repetida ejecución del cuerpo mientras la guardia sea verdadera. Al igual que las expresiones condicionales, debido a la guardia tendremos dos reglas de evaluación.

Si la guardia es falsa, el cuerpo del ciclo no se evalúa y el ciclo se comporta como el comando **skip**, esto es, no hace nada

$$\frac{B(b,s) = ff}{\langle \textbf{while } b \textbf{ do } C, s\rangle \longrightarrow s}\ \textsc{While-Eval-Ciclo}_{ff}$$

En un estado s, si la guardia es verdadera, el cuerpo C se evalúa en s y si termina en un estado s', en este nuevo estado se repite la evaluación del ciclo

$$\frac{B(b,s) = tt \qquad \langle C, s\rangle \longrightarrow s' \qquad \langle \textbf{while } b \textbf{ do } C, s'\rangle \longrightarrow s''}{\langle \textbf{while } b \textbf{ do } C, s\rangle \longrightarrow s''} \text{ While-Eval-Ciclo}_{tt}$$

Para ver esta regla de una manera más intuitiva podemos pensar en el ciclo desdoblado; una vez que se ha determinado que la guardia es verdadera podemos ver la evaluación que sigue como

$$C; \textbf{while } b \textbf{ do } C$$

entonces solo debemos recordar cómo se ve la evaluación de una secuencia de comandos While-Eval-Sec para obtener la regla While-Eval-Ciclo$_{tt}$.

1.3. Lógica de Hoare, semántica axiomática

Como ya se mencionó, la lógica de Hoare es una semántica axiomática, es un sistema de inferencia que consiste de una serie de axiomas y reglas que permiten razonar sobre programas While, es decir, escribir propiedades de interés que se espera un programa tenga y demostrar mediante razonamiento deductivo que las poseen o satisfacen. Con esto vemos que la lógica de Hoare es una herramienta que nos permite llevar a cabo verificación de programas: proporcionar especificaciones y demostrar formalmente la corrección de un programa con respecto a su especificación.

Para este fin vamos a anotar los programas (su texto) con aserciones, éstas son predicados lógicos que afirman propiedades sobre los programas mediante relaciones entre los valores de las variables en un punto dado de la ejecución. Puesto de otra forma, las aserciones nos permiten describir estáticamente el comportamiento dinámico de un programa.

Los axiomas y reglas de inferencia de la lógica nos van a decir entonces cómo demostrar que lo que se está afirmando sobre un programa mediante aserciones es cierto analizando los comandos de los que se compone el programa y haciendo uso de las propiedades de estos últimos. En este sentido la lógica de Hoare es *composicional*: la demostración de una propiedad de interés de un comando va a depender de qué comando es y de las propiedades de los posibles comandos que lo componen, esto es, depende de su estructura y de sus subexpresiones como una expresión del lenguaje While.

Las aserciones expresan relaciones entre variables, y debido a que los valores de las variables durante la ejecución de un programa están dados por el estado, consideraremos las aserciones como predicados lógicos sobre el espacio S de estados. El lector puede consultar una definición

formal del lenguaje de aserciones y su semántica en los siguientes textos [20, 6], a continuación sólo daremos las nociones básicas.

Podemos ver las aserciones como fórmulas en un lenguaje de primer orden con igualdad, con dominio los números enteros, y operadores aritméticos, lógicos y desigualdad con su interpretación estándar, cuyas variables coinciden con las variables de While. Dada una aserción P, decimos que un estado s *satisface* P, lo cual escribimos Ps[2], si y solo si $\mathcal{M} \models \mathcal{P}$, con $\mathcal{M}$ el modelo del lenguaje, y $\mathcal{P}$ se obtiene de reemplazar en P las variables libres con su respectivo valor según s.

Entonces una anotación para un comando C de While es una terna

$$\{P\}\ C\ \{Q\}$$

donde P y Q son aserciones. Estas ternas son denominadas *Ternas de Hoare*[3], P es la *precondición* y Q es la *poscondición*. Intuitivamente, queremos que la anotación describa el comportamiento del comando, esto es, las aserciones P y Q reflejan el cambio inducido sobre el estado, efecto de la ejecución de C. Para lograr esto debemos analizar la estructura de C: qué comando es, qué ocurre durante su ejecución y qué se puede decir de los comandos que lo componen sintácticamente (cuando estos existan).

En una terna de Hoare, la precondición P nos dice qué ocurre en el punto de ejecución inmediatamente antes de C, y la poscondición Q nos dice qué ocurre inmediatamente después de C. Así, informalmente, el significado de una terna de Hoare $\{P\}\ C\ \{Q\}$ es "si el comando C se ejecuta en un estado que satisface la precondición P y su ejecución termina, entonces el estado resultante satisface la poscondición Q".

Nótese que las ternas de Hoare definidas describen propiedades para un comando bajo el supuesto que la ejecución del mismo termina, esto significa que la lógica aquí presentada es una lógica de *corrección parcial*, los axiomas y reglas de inferencia buscan capturar el comportamiento general de los comandos pero no ofrecen forma alguna de asegurar que el comando termina. Una lógica que además asegura que la ejecución de un programa termina es una lógica de *corrección total* y es posible extender la lógica de Hoare con tal finalidad [1]. En el presente trabajo solo nos enfocaremos en la corrección parcial de programas.

El significado formal de una terna de Hoare lo podemos escribir apoyándonos de la relación de evaluación para los comandos previamente definida

$$\{P\}\ C\ \{Q\} \equiv \forall s\ s'.\ \langle C, s\rangle \longrightarrow s' \to Ps \to Qs' \tag{1.1}$$

Las ternas de Hoare serán las fórmulas o juicios del sistema de inferencia. La meta ahora es

[2]Aquí adoptamos la notación funcional sin paréntesis para $P(s)$. En la literatura se suele escribir también $s \vDash P$.
[3]Hoare usó en su artículo [11] una notación distinta: $P\ \{C\}\ Q$. Esta notación es ahora obsoleta.

proponer axiomas y reglas que capturen el efecto de la ejecución de cada comando (permaneciendo fiel a la relación descrita por la semántica operacional) de tal forma que la verificación de programas pueda llevarse a cabo mediante razonamiento deductivo.

El caso del comando **skip** es bastante sencillo pues este comando no produce ningún efecto sobre el estado, por lo que si sabemos que P es cierta antes de ejecutar **skip**, P sigue siendo cierta al terminar

$$\frac{}{\{P\}\ \textbf{skip}\ \{P\}}\ \textsc{While-Hoare-Skip}$$

Es en la asignación de variables cuando ocurren modificaciones al estado, pero el cambio ocurre para una variable en específico: la que se está asignando.

Si queremos anotar una asignación de variable $x := a$ con una poscondición Q, donde Q es tentativamente un predicado que busca describir una relación entre el "nuevo" valor de x y algún otro número (por ejemplo, el valor de otra variable), Q se debe satisfacer en el nuevo estado con el nuevo valor de x, pero este valor es el resultado de evaluar la expresión aritmética a en el estado previo a la asignación (y que contiene el valor "viejo" de x). Dicho de otra forma, para concluir Q para x después de la asignación, debe ser el caso que Q es cierta para $A(a, s)$, con s el estado previo a la asignación. Abusando de notación, esto último lo escribimos $Q[A(a, s)/x]$[4] y tenemos el axioma

$$\frac{}{\{Q[A(a,s)/x]\}\ x := a\ \{Q\}}\ \textsc{While-Hoare-Asig}$$

Notemos aquí que Q puede mencionar variables distintas de x pues sus valores no cambian después de la asignación.

Suele ser común al encontrar por primera vez el axioma While-Hoare-Asig pensar que la precondición y la poscondición están al revés, que se debe reemplazar x por $A(a, s)$ en la aserción Q tras la asignación puesto que ese es el nuevo valor de x. Esto es incorrecto. Pensemos por un momento que el axioma para asignación es

$$\frac{}{\{Q\}\ x := a\ \{Q[A(a,s)/x]\}}$$

podemos ver fácilmente el error en el axioma considerando un ejemplo muy sencillo: sea a la expresión $x + 1$ y Q el predicado $x = n$, donde n es el valor de x en el ambiente s previo a la asignación. En la terna

$$\{x = n\}\ x := x + 1\ \{(x = n)[A(x + 1, s)/x]\}$$

[4]Recordemos que las aserciones son predicados sobre los estados, y el estado resultado de la ejecución de la asignación es $s[A(a, s)/x]$ según la regla While-Eval-Asig.

la precondición es claramente cierta, pero la poscondición es falsa pues $A(x+1, s) = s(x)+1 = n+1$, con lo que la poscondición nos queda $n+1 = n$, y esto es falso para cualquier número entero. El error radica en el hecho que si la aserción Q es cierta para el valor particular de x previo a la asignación, puede ya no ser cierta una vez que el valor de x cambia tras la asignación.

Para una secuencia de comandos $C_1; C_2$ debemos analizar qué pasa con los comandos C_1 y C_2. Si queremos anotar el comando $C_1; C_2$ con una precondición P y una poscondición Q, la ejecución del comando debe iniciar en un estado que satisface P y de terminar, lo hace en un estado que satisface Q, por lo que la ejecución de C_1 debe iniciar en un estado que satisface P y posteriormente la ejecución de C_2 terminar en algún estado que satisface Q. Recordando la regla While-Eval-Sec, la ejecución de C_2 continúa en el estado en que terminó C_1, así, si podemos asegurar que este estado satisface una poscondición R para C_1, entonces R funciona también como precondición para C_2. Con esto podemos "descomponer" una terna del comando $C_1; C_2$ en dos ternas para los comandos C_1 y C_2

$$\frac{\{P\}\ C_1\ \{R\} \qquad \{R\}\ C_2\ \{Q\}}{\{P\}\ C_1; C_2\ \{Q\}}\ \textsc{While-Hoare-Sec}$$

Para anotar una expresión condicional **if** b **then** C_1 **else** C_2 con una poscondición Q debemos tener en cuenta que Q va a funcionar como poscondición tanto para C_1 como para C_2, pues recordando las reglas While-Eval-Cond$_{tt}$ y While-Eval-Cond$_{ff}$, el estado final para la expresión condicional es el estado final de C_1 o C_2 dependiendo el valor de la guardia.

En cuanto a la precondición, si la expresión condicional se ejecuta en un estado que satisface P, primero se determina el valor de la guardia y continúa en ese estado la ejecución de C_1 o C_2, y en ese momento sabemos dos cosas: que se satisface P (pues el estado es el mismo) y que la guardia es verdadera o falsa en ese estado según sea el caso. Así podemos anotar C_1 con la precondición $P \wedge B(b, s) = tt$ y C_2 con la precondición $P \wedge B(b, s) = ff$.

$$\frac{\{P \wedge b\}\ C_1\ \{Q\} \qquad \{P \wedge \sim b\}\ C_2\ \{Q\}}{\{P\}\ \textbf{if}\ b\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \{Q\}}\ \textsc{While-Hoare-Cond}$$

(Abusando de notación escribimos b cuando la guardia es *tt* y $\sim b$ cuando la guardia es *ff*.)

Vale la pena analizar detenidamente la regla para ciclos, siendo los ciclos de gran importancia para la programación imperativa.

Primero pensemos en un ciclo cuya guardia siempre es falsa. La ejecución del ciclo empieza con la verificación de la guardia, como esta es falsa el cuerpo del ciclo no se ejecuta y el comando se comporta como **skip**. La regla While-Hoare-Skip nos dice que la precondición es también la poscondición pues no hay cambio alguno al estado, entonces podríamos pensar que

la verificación del ciclo nos llevaría a una terna de la forma

$$\frac{}{\{P\}\ \textbf{while}\ b\ \textbf{do}\ C\ \{P\}}$$

pero también sabemos que en el estado en que se ejecutó el ciclo la guardia es falsa, por lo que al terminar no solo sabemos P, también sabemos que $B(b, s) = ff$ y podemos agregar esta información a la poscondición

$$\frac{}{\{P\}\ \textbf{while}\ b\ \textbf{do}\ C\ \{P \wedge \sim b\}}$$

Más aún, sin importar la guardia, siempre podemos agregar $\sim b$ a la poscondición del ciclo ya que este termina cuando la guardia deja de ser verdadera, y si nunca termina podemos concluir cualquier cosa, en particular $\sim b$ (recordar el significado de una terna, ecuación 1.1).

Vamos a querer mantener la forma de esta terna al analizar el caso en que la guardia es verdadera para capturar ambos comportamientos en una sola regla.

Ahora pensemos en un ciclo cuyo cuerpo se ejecuta al menos una vez. La primera vez que se verifica que la guardia es verdadera se ejecuta el cuerpo, y si termina se ejecuta de nuevo el ciclo. Pensemos de nuevo en el ciclo desdoblado C; **while** b **do** C, de la regla While-Hoare-Sec sabemos que la poscondición de C es la precondición de **while** b **do** C, esto es, del ciclo mismo. También sabemos que al iniciar la ejecución de C (en cualquier iteración) la guardia era verdadera. Entonces podríamos tratar de anotar los comandos que componen el ciclo desdoblado y llegar a una regla de la forma

$$\frac{\{P \wedge b\}\ C\ \{Q\} \qquad \{Q\}\ \textbf{while}\ b\ \textbf{do}\ C\ \{P \wedge \sim b\}}{\{P\}\ \textbf{while}\ b\ \textbf{do}\ C\ \{P \wedge \sim b\}}$$

Pero si continuamos desdoblando el ciclo y aplicando la regla de secuencia podemos ver que se acumularían ternas sobre C con diferentes precondiciones y poscondiciones, por lo cual no es óptimo desdoblar el ciclo ya que siempre obtendremos una nueva terna para el ciclo mismo. Por ello queremos que la regla para ciclos capture el comportamiento de cualquier número de iteraciones del cuerpo C, esto es, queremos anotar C de tal forma que podamos concluir una poscondición que sea cierta al final de cada iteración.

Este razonamiento sugiere que la poscondición para C debe ser, o al menos es deseable que sea P: al final de la última iteración del cuerpo la poscondición P nos permitiría concluir $P \wedge \sim b$ para el ciclo en su totalidad, y al final de una iteración "intermedia" nos permitiría mantener P como precondición de C para la próxima iteración tal como se tiene en la primera iteración del ciclo.

Si podemos encontrar una aserción P (para cada ciclo en particular) que se conserve entre iteraciones, esto es, sea verdadera al iniciar y terminar cada ejecución del cuerpo C, con una

sola terna podemos capturar todas las ternas que surgirían como resultado de desdoblar el ciclo como se estaba explicando previamente y podemos concluir P como poscondición del ciclo fácilmente. Así tenemos la regla

$$\frac{\{P \wedge b\}\ C\ \{P\}}{\{P\}\ \textbf{while}\ b\ \textbf{do}\ C\ \{P \wedge \sim b\}}\ \textsc{While-Hoare-Ciclo}$$

A esta aserción P se le conoce como el *invariante del ciclo*. Es fácil ver que el invariante es una aserción que se mantiene verdadera tras cualquier número de iteraciones del ciclo, incluso en el primer caso considerado donde la guardia siempre es falsa y no se ejecuta el cuerpo del ciclo ni una sola vez.

Hasta este punto tenemos una regla para cada construcción sintáctica de While, decimos entonces que la lógica es *dirigida por la sintaxis*, en el sentido que hay una clara correspondencia entre las producciones de la gramática del lenguaje y las reglas semánticas, que son los axiomas y reglas de inferencia. Sin embargo, la forma de estas reglas restringe los escenarios en que pueden ser aplicadas, por ejemplo, no siempre tenemos un invariante presente en la precondición y poscondición al llegar a un ciclo dentro del texto de un programa más grande.

La regla de consecuencia es de gran utilidad pues permite cambiar las precondiciones y poscondiciones de una terna por otras más apropiadas que permitan aplicar alguna otra regla; dicho de otra forma, esta regla permite "adaptar" una terna en el contexto de prueba actual a la forma de una terna presente en la conclusión de alguna otra regla de la lógica.

La regla de consecuencia nos permitirá cambiar la precondición por una más fuerte o la poscondición por una más débil. Si tenemos una terna para un programa con precondición P' y poscondición Q', entonces cualquier predicado que sea consecuencia lógica de Q' funcionará como poscondición, y cualquier predicado que implique a P' funcionará como precondición. Como la nueva precondición implica P', esta es más fuerte. Como la nueva poscondición es consecuencia de Q', esta es más débil.

$$\frac{P \to P' \qquad \{P'\}\ C\ \{Q'\} \qquad Q' \to Q}{\{P\}\ C\ \{Q\}}\ \textsc{While-Hoare-Consec}$$

En la figura 1.2 se encuentran listados los axiomas y reglas de inferencia que se discutieron en esta sección, en conjunto conforman la lógica de Hoare para While.

La lógica de Hoare le da un significado a las construcciones sintácticas de While de una manera distinta a como se logra con las otras semánticas. La semántica operacional le proporciona un significado a un programa individual describiendo ya sea la secuencia de cómputos que induce en una máquina (paso pequeño) o la relación entre el estado inicial y el final (paso grande), la semántica denotativa lo hace mediante denotaciones que abstraen el efecto de la

$$\frac{}{\{P\}\ \textbf{skip}\ \{P\}}\ \textsc{While-Hoare-Skip}$$

$$\frac{}{\{Q[A(a,s)/x]\}\ x := a\ \{Q\}}\ \textsc{While-Hoare-Asig}$$

$$\frac{\{P\}\ C_1\ \{R\} \qquad \{R\}\ C_2\ \{Q\}}{\{P\}\ C_1; C_2\ \{Q\}}\ \textsc{While-Hoare-Sec}$$

$$\frac{\{P \wedge b\}\ C_1\ \{Q\} \qquad \{P \wedge \sim b\}\ C_2\ \{Q\}}{\{P\}\ \textbf{if}\ b\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \{Q\}}\ \textsc{While-Hoare-Cond}$$

$$\frac{\{P \wedge b\}\ C\ \{P\}}{\{P\}\ \textbf{while}\ b\ \textbf{do}\ C\ \{P \wedge \sim b\}}\ \textsc{While-Hoare-Ciclo}$$

$$\frac{P \to P' \qquad \{P'\}\ C\ \{Q'\} \qquad Q' \to Q}{\{P\}\ C\ \{Q\}}\ \textsc{While-Hoare-Consec}$$

Figura 1.2: La lógica de Hoare para el lenguaje While

ejecución, pero la lógica de Hoare por su parte define un sistema de inferencia que nos permite razonar sobre los programas, demostrar propiedades y verificar corrección parcial de manera composicional, entonces el significado de un programa es todo aquello que puede demostrarse en la lógica. Floyd [10] y Hoare [11] notan que la especificación de métodos de prueba proporcionan una definición formal del significado del lenguaje; en nuestro caso, los axiomas y reglas de inferencia se encargan de proporcionar el significado de cada construcción del lenguaje y explican cómo razonar con ellas.

Una aclaración importante que debe hacerse aquí es que la lógica de Hoare, siendo un sistema de inferencia, conlleva dos nociones: demostrabilidad[5] y verdad.

La semántica de una terna de Hoare se dio haciendo uso de la semántica operacional definida en la sección 1.2, la verdad de estos juicios está dada por la verdad de la fórmula en la ecuación 1.1. Cuando una terna de Hoare es verdadera con respecto a esta semántica lo escribimos

$$\vDash \{P\}\ C\ \{Q\}$$

Cuando una terna se puede demostrar, probar o derivar en la lógica, partiendo de los axiomas y mediante la aplicación de reglas de inferencia (que es una acción puramente sintáctica), lo

[5]En inglés el término es *provability*.

escribimos

$$\vdash \{P\}\ C\ \{Q\}$$

Idealmente estas dos nociones coinciden, lo cual expresamos con las propiedades

- *Solidez*[6] *(soundness)*: la lógica es sólida si toda terna que se pueda derivar en la lógica es verdadera (con respecto a la semántica de las ternas), es decir

$$\vdash \{P\}\ C\ \{Q\} \quad \text{implica} \quad \vDash \{P\}\ C\ \{Q\}$$

- *Completitud (completeness)*: la lógica es completa si toda terna que es verdadera (con respecto a la semántica) se puede derivar en la lógica, es decir

$$\vDash \{P\}\ C\ \{Q\} \quad \text{implica} \quad \vdash \{P\}\ C\ \{Q\}$$

La lógica de Hoare del lenguaje While es sólida y completa (aunque con algunas consideraciones [6]). Una demostración de este resultado se puede encontrar en el libro de Nielson [15].

En resumen, en este capítulo se introdujo el lenguaje While, presentamos su gramática (figura 1.1), definimos los estados y las funciones A y B para evaluación de expresiones aritméticas y booleanas respectivamente, para después definir la semántica operacional mediante la relación $\longrightarrow$ (sección 1.2), entonces definimos formalmente las ternas de Hoare (ecuación 1.1) apoyándonos de la semántica operacional y finalmente presentamos los axiomas y reglas de inferencia de la lógica de Hoare (figura 1.2), explicando la idea detrás de cada regla basándonos en lo que ya sabemos ocurre durante la ejecución de programas. En el siguiente capítulo seguiremos el mismo camino para diseñar e implementar una semántica axiomática para programas funcionales.

[6] En algunos textos también se traduce como *corrección*.

Capítulo 2

Lógica de Hoare para un núcleo funcional simple

En el capítulo anterior se presentó la ya bien conocida y estudiada lógica de Hoare para el lenguaje While. En este capítulo nos ocuparemos de desarrollar una semántica axiomática al estilo de la lógica de Hoare imperativa pero para un lenguaje funcional que cuenta con operaciones básicas sobre números naturales (sucesor, predecesor, suma y producto) y operadores booleanos (comparación de números naturales, negación y conjunción).

Este lenguaje será bastante simple, cuenta con las estructuras de control comúnmente encontradas en lenguajes de programación funcionales modernos (Haskell y OCaml por nombrar un par de ejemplos) como son las expresiones condicionales o if, declaración local de variables o let y la aplicación de funciones. Por todo esto lo apodamos *lenguaje funcional simple*, su implementación, que consiste de la gramática, las reglas de evaluación y la semántica axiomática, se realizará en el asistente de pruebas Coq.

El desarrollo de la semántica axiomática que se presentará en este capítulo está inspirado en gran medida en los artículos [18, 13]. En ambos trabajos se presenta un lenguaje funcional con diversas características avanzadas como son tipos de datos algebraicos, polimorfismo y tipos recursivos, también se introduce un lenguaje de orden superior tipificado para las aserciones que requiere la semántica axiomática estilo lógica de Hoare para anotar y verificar las construcciones de dicho lenguaje. En este capítulo (y en esta tesis) no buscamos replicar en su totalidad los trabajos citados, pero sí recuperar las nociones básicas que subyacen el razonamiento informal que se sigue al trabajar con programas funcionales sencillos, por ello nos restringimos a explorar las estructuras de control básicas del paradigma funcional y diseñar axiomas y reglas que capturen adecuadamente su significado, no en una lógica definida específicamente para el lenguaje funcional simple, sino en la lógica de Coq, que nos proporciona una lógica de orden

superior al mismo tiempo que el medio para implementar nuestro lenguaje.

Pero primero, estableceremos una serie de restricciones sobre nuestro lenguaje con el fin de tener una implementación sencilla y así poder enfocarnos en el análisis de las construcciones de este y sus características, y finalmente, en el desarrollo e implementación de las aserciones y reglas de inferencia de la lógica:

- La estrategia de evaluación será llamada por valor, donde los números naturales serán los únicos valores para este lenguaje.
- Toda expresión booleana será utilizada únicamente en las guardias de una expresión condicional.
- Habrá dos conjuntos (no necesariamente ajenos) de nombres o identificadores, uno para nombres de variables y otro para nombres de funciones. En la gramática del lenguaje no habrá lugar para confusión, será obvio cuándo un identificador se refiere a una variable y cuándo se refiere a una función.
- Toda función tomará un valor como argumento y regresará un valor como resultado en caso de terminar su ejecución.
- Además, toda función debe tener un nombre y ser definida fuera del texto de un programa. En este lenguaje las funciones no son valores, no se permiten abstracciones lambda, no pueden ser pasadas como parámetros en una aplicación o ser el resultado de otra, y no se pueden ligar al declarar una variable local.
- Toda función debe ser cerrada, esto es, el cuerpo de la función (que es un programa) solo puede hablar del parámetro de la función, salvo por las variables que sean declaradas localmente dentro del mismo.
- Se permite aplicar otras funciones dentro de una función (incluso ella misma).
- La evaluación de programas requiere de dos ambientes distintos:
    - Un ambiente de variables que será una función de las variables en los números naturales con los valores definidos para cada variable en un punto dado de (la ejecución de) un programa. Por comodidad el ambiente será una función total, el valor "por defecto" de una variable que no ha sido explícitamente declarada será 0.
    - Un ambiente de funciones con las definiciones de funciones necesarias para la ejecución de un programa. Este ambiente es distinto al ambiente de las variables y es fijo durante toda la ejecución. Toda función que vaya a usarse en un programa debe estar previamente definida en el ambiente.

El propósito del ambiente de variables es almacenar el valor de una variable y poder recuperarlo durante la ejecución para evitar realizar sustituciones sintácticas y generar código "al vuelo", además de evitar realizar modificaciones al texto de un programa cuando una sustitución no está bien definida, como renombrar variables en busca de expresiones que sean α-equivalentes.

2.1. El asistente de pruebas Coq

El asistente de pruebas Coq [7] proporciona un marco de trabajo, con un lenguaje de programación funcional y una lógica de orden superior muy expresiva, en el cual es posible enunciar teoremas matemáticos y especificaciones de software, y demostrar estos enunciados. Famosas aplicaciones de Coq incluyen la demostración del teorema de los cuatro colores en teoría de gráficas[1] y el proyecto CompCert[2] para certificación de compiladores útiles en aplicaciones críticas.

En Coq se pueden formalizar teorías y razonamiento matemático definiendo objetos, tipos de datos, algoritmos (o programas), predicados lógicos, y realizando pruebas formales en el sistema formal subyacente usando el lenguaje Gallina[3], donde se puede expresar de manera detallada y precisa hechos matemáticos o especificaciones de programas, y construir interactivamente demostraciones formales y rigurosas.

Internamente Coq se compone de un kernel con un número relativamente pequeño de construcciones sintácticas, reglas para verificación de tipos y cómputos; y de un ambiente para construir *términos de prueba*, que son términos de Gallina y que serán verificados por el kernel.

En Coq, una prueba se construye interactivamente mediante aplicación de *tácticas* utilizando la estrategia de *razonamiento hacia atrás*, esto es, cuando se tiene una proposición que se quiere demostrar, la *meta*, el usuario parte de esta proposición y mediante el uso de tácticas transforma la meta en una serie de submetas tal que resolver las submetas implica resolver la meta original. Una demostración termina cuando no quedan submetas por resolver. Para ejemplificar consideremos la regla para introducción de la conjunción en la lógica proposicional

$$\frac{A \qquad B}{A \wedge B} \wedge\text{I}$$

Esta regla se puede leer como "para demostrar $A \wedge B$ es suficiente demostrar A y demostrar B", y la táctica `split` hace justamente eso, transforma una meta de la forma $A \wedge B$ en dos submetas

[1] Incluida en la biblioteca *Mathematical Components*. dirección: `https://math-comp.github.io/`

[2] `https://compcert.org/`

[3] Una implementación y extensión del cálculo de construcciones inductivas, abreviado CIC por sus siglas en inglés *Calculus of Inductive Constructions*.

por resolver A y B.

Las tácticas son los pasos de deducción que guían la construcción de un término de prueba de una proposición dada P, el cual es un término de Gallina con tipo P, siguiendo la correspondencia Curry-Howard. Así, los juicios lógicos en Coq son juicios de tipificado: el kernel verifica que el término de prueba p generado es en efecto una prueba de P bajo un contexto de prueba Γ con hipótesis locales, esto es, verifica el secuente $\Gamma \vdash P$ al verificar el juicio de tipificado $\Gamma \vdash p : P$.

Como lenguaje de programación, Coq no es de propósito general (a diferencia de C o Haskell), programas escritos en Coq son funciones totales y su ejecución termina. Por otro lado su sistema de tipos es muy rico y ofrece polimorfismo, operadores de tipos y tipos dependientes, así como tipos de datos inductivos y otras estructuras más sofisticadas. Con todo esto Coq se puede aprovechar como un ambiente para verificación formal y certificación de software en el cual se puede especificar el comportamiento de programas, demostrar la corrección de estos con respecto a su especificación y extraer software que es correcto por construcción.

2.2. Gramática del lenguaje funcional simple

Por simplicidad, asumimos que el conjunto de nombres de variables, denotado id, es el mismo conjunto para nombres de funciones. También, los números naturales en este lenguaje se implementan con los números del tipo `nat` en Coq, mientras que las constantes booleanas son de tipo `bool`.

La gramática del lenguaje funcional simple se da en notación estilo BNF en la figura 2.1. En una declaración de función $f\ x \Rightarrow P$, el primer identificador f es el nombre de la función, el segundo x es el nombre del parámetro (una variable) y P es el cuerpo de la función.

Esta gramática y la definición de evaluación de paso grande definida a continuación obedecen las restricciones listadas previamente además de cuidar los tipos del lenguaje eliminando la necesidad de definir reglas de tipificado. Tanto P (programas) como A (expresiones aritméticas) tienen tipo `nat`, mientras que B (expresiones booleanas) tiene tipo `bool`. Toda función tiene tipo $\texttt{nat} \to \texttt{nat}$. No se pueden confundir funciones con variables pues los nombres de funciones sólo figuran en la construcción sintáctica para la aplicación de función, además las definiciones de variables y funciones se encuentran en ambientes distintos durante la ejecución.

2.3. Semántica operacional de paso grande

La evaluación de programas del lenguaje funcional simple está dada por una semántica operacional de paso grande o natural, para este trabajo no nos interesa la secuencia de reducciones

			Programas
P	$::=$	A	*Expresión aritmética*
	$\mid$	**if** B **then** P_1 **else** P_2	*Condicional*
	$\mid$	**let** $x = P_1$ **in** P_2	*Declaración local*
	$\mid$	$f\ P$	*Aplicación de función*
			Expresiones aritméticas
A	$::=$	n	*Número natural*
	$\mid$	x	*Variable*
	$\mid$	**succ** P	*Sucesor*
	$\mid$	**pred** P	*Predecesor*
	$\mid$	$P_1 + P_2$	*Suma*
	$\mid$	$P_1 * P_2$	*Multiplicación*
			Expresiones booleanas
B	$::=$	b	*Constante booleana*
	$\mid$	$P_1 = P_2$	*Igualdad*
	$\mid$	$P_1 < P_2$	*Menor que*
	$\mid$	**not** B	*Negación*
	$\mid$	B_1 **and** B_2	*Conjunción*
			Funciones
F	$::=$	$f\ x \Rightarrow P$	*Declaración de función*

Figura 2.1: El lenguaje funcional simple (sintaxis)

que sigue un programa como lo ejecutaría una computadora en el bajo nivel o como lo describe una semántica de paso pequeño, sólo nos interesa el valor al que se evaluará un programa para razonar con ese valor, de manera análoga a la lógica de Hoare para WHILE donde nos importa el estado final al que llega un programa al terminar su ejecución.

La semántica operacional estará dada mediante una relación de evaluación estándar cuya definición depende de dos ambientes de acuerdo con las restricciones previamente listadas.

El primer ambiente es el de variables, que es una función total $id \to$ `nat`. Este ambiente de variables tiene un rol similar al estado en el lenguaje WHILE: dada una variable x y un ambiente a, el valor de x en a es $a(x)$, y como se mencionó en las restricciones del lenguaje, toda variable que no ha sido declarada explícitamente en el texto de un programa tiene valor 0.

El segundo ambiente es el de funciones y es una colección finita de pares $(f, f\ x \Rightarrow p)$ donde la primera entrada es el nombre de la función y la segunda es la definición de la función asociada a ese nombre. Este ambiente de funciones está presente de manera implícita en la definición de la relación de evaluación, nunca cambia durante la ejecución de un programa, y están definidas de manera única las funciones necesarias para la evaluación de un programa, es decir, siempre que haya una aplicación de función en el texto de un programa hay una única función definida en el ambiente de funciones con nombre el identificador escrito en la aplicación.

Este ambiente está implementado como una función parcial usando el tipo `option` de Coq

$$ambiente_F : id \to \texttt{option}\ F$$

donde un par $(f, f\ x \Rightarrow p)$ está en el ambiente si y sólo si

$$ambiente_F(f) = \texttt{Some}\ (f\ x \Rightarrow p) \tag{2.1}$$

De aquí en adelante adoptamos la siguiente convención para referirnos a las construcciones sintácticas del lenguaje:

- Las metavariables p, q representan programas, e y g representan expresiones aritméticas y booleanas respectivamente; se usarán subíndices cuando sea pertinente.
- La metavariable a es el ambiente de variables.
- Para las variables (numéricas) usamos las letras x, y, z, ..., y la letra f para función.
- Para constantes numéricas usamos las letras n, m, ..., para constantes booleanas usamos la letra b.

En cuanto al ambiente de funciones, este no será nombrado explícitamente, cuando se requiera expresar que una función debe estar definida se escribirá su definición usando metavariables como se espera esté definida en el ambiente en vez de escribir de nuevo la ecuación 2.1.

2.3.1. Evaluación de programas *P*

La relación de evaluación para programas, expresiones aritméticas y expresiones booleanas están mutuamente definidas por lo que las tres relaciones están implementadas como tipos inductivos mutuamente definidos en Coq. La relación

$$p/a \Downarrow n$$

se lee como "el programa p en el ambiente a se evalúa a n". Las reglas de evaluación para programas se presentan en la figura 2.2.

$$\frac{e/a \Downarrow n}{p/a \Downarrow n}\ \text{PEA} \qquad \frac{g/a \Downarrow \mathit{tt} \qquad p_1/a \Downarrow n}{\textbf{if } g \textbf{ then } p_1 \textbf{ else } p_2/a \Downarrow n}\ \text{PIf}_{\text{True}}$$

$$\frac{g/a \Downarrow \mathit{ff} \qquad p_2/a \Downarrow n}{\textbf{if } g \textbf{ then } p_1 \textbf{ else } p_2/a \Downarrow n}\ \text{PIf}_{\text{False}} \qquad \frac{p_1/a \Downarrow n \qquad p_2/a[x \to n] \Downarrow m}{\textbf{let } x = p_1 \textbf{ in } p_2/a \Downarrow m}\ \text{PLet}$$

$$\frac{f\ x \Rightarrow p \qquad \mathit{cerrada}\ f \qquad q/a \Downarrow n \qquad p/[x \to n] \Downarrow m}{f\ q/a \Downarrow m}\ \text{PApp}$$

Figura 2.2: Relación de evaluación para programas

La regla PEA aplica al caso en que el programa p es la expresión aritmética e, correspondiente a la primera regla de producción en la gramática de programas (símbolo gramatical P en la figura 2.1).

En la regla PLET, el ambiente $a[x \to n]$ se define como

$$a[x \to n](y) = \begin{cases} n & \text{si } y = x \\ a(y) & \text{e.o.c.} \end{cases}$$

la variable x sólo existe dentro del cuerpo p_2, su alcance, y es en la ejecución del cuerpo que el ambiente de variables cambia; en este nuevo ambiente la variable x tiene ligado el valor resultado de evaluar p_1 en el ambiente original. Una vez que termina la ejecución de p_2 regresamos al ambiente original y el resultado de p_2 es el resultado de la expresión let en su totalidad.

Si ya existía una variable con el mismo nombre que la variable que se está declarando, esta última "reemplaza" a la anterior (y en consecuencia el valor en el ambiente ligado a esa variable) pero sólo dentro de su alcance; la variable previa y su valor se recuperan una vez salimos del cuerpo p_2 del let.

En la regla PAPP, el ambiente $[x \to n]$ se define como

$$[x \to n](y) = \begin{cases} n & \text{si } y = x \\ 0 & \text{e.o.c.} \end{cases}$$

Dado que las funciones son cerradas, los cuerpos sólo pueden referirse a su parámetro (salvo por declaraciones locales que existan dentro de este), la ejecución (del cuerpo) de la función ocurre en un ambiente donde sólo existe el argumento que recibió en la aplicación. La hipótesis $\mathit{cerrada}\ f$ nos asegura que la función f es una función cerrada del lenguaje.

El predicado $cerrada$ se define como

$$cerrada\ (f\ x \Rightarrow p) \equiv \forall y.\ y \in fv(p) \rightarrow x = y \tag{2.2}$$

donde $fv(p)$ es el conjunto de variables libres en el cuerpo de la función f. Por comodidad escribiremos solo el nombre de la función en vez de su definición cuando solicitemos que una función sea cerrada, tal y como se escribió en la regla PApp. El lector debe recordar que $cerrada$ es un predicado sobre definiciones de funciones, no sobre nombres o identificadores de funciones.

Veamos un par de ejemplos. La función definida como $f\ x \Rightarrow x * x$ es cerrada, el conjunto de variables libres en su cuerpo es $\{x\}$, que solo contiene al parámetro x de la función. Por otro lado, la función definida como $g\ y \Rightarrow z + y$ no es cerrada, el conjunto de variables libres en su cuerpo es $\{z, y\}$, con z una variable libre distinta del parámetro y.

2.3.2. Evaluación de expresiones aritméticas *A*

Las reglas de evaluación para expresiones aritméticas se presentan en la figura 2.3. La relación

$$e/a \Downarrow n$$

se lee "la expresión aritmética e en el ambiente a se evalúa a n".

$$\frac{}{n/a \Downarrow n}\ \textsc{ANum} \qquad \frac{}{x/a \Downarrow a(x)}\ \textsc{AId} \qquad \frac{p/a \Downarrow n}{\mathbf{succ}\ p/a \Downarrow S\ n}\ \textsc{ASucc}$$

$$\frac{p/a \Downarrow n}{\mathbf{pred}\ p/a \Downarrow pred\ n}\ \textsc{APred} \qquad \frac{p_1/a \Downarrow n \qquad p_2/a \Downarrow m}{p_1 + p_2/a \Downarrow n + m}\ \textsc{APlus}$$

$$\frac{p_1/a \Downarrow n \qquad p_2/a \Downarrow m}{p_1 * p_2/a \Downarrow n * m}\ \textsc{AProd}$$

Figura 2.3: Relación de evaluación para expresiones aritméticas

En la regla APred, $pred$ es la función predecesor sobre los números naturales definida como

$$pred(n) = \begin{cases} 0 & \text{si } n = 0 \\ m & \text{si } n = S\ m \end{cases}$$

2.3.3. Evaluación de expresiones booleanas *B*

Las reglas de evaluación para expresiones booleanas se presentan en la figura 2.4. La relación

$$g/a \Downarrow b$$

se lee "la expresión booleana g en el ambiente a se evalúa a b".

$$\frac{}{b/a \Downarrow b}\ \textsc{BBool} \qquad \frac{p_1/a \Downarrow n \qquad p_2/a \Downarrow m}{p_1 = p_2/a \Downarrow n =? m}\ \textsc{BEq} \qquad \frac{p_1/a \Downarrow n \qquad p_2/a \Downarrow m}{p_1 < p_2/a \Downarrow n <? m}\ \textsc{BLt}$$

$$\frac{g/a \Downarrow b}{\textbf{not}\ g/a \Downarrow \sim b}\ \textsc{BNot} \qquad \frac{g_1/a \Downarrow b_1 \qquad g_2/a \Downarrow b_2}{g_1\ \textbf{and}\ g_2/a \Downarrow b_1\ \&\ b_2}\ \textsc{BAnd}$$

Figura 2.4: Relación de evaluación para expresiones booleanas

En la regla BLt, el operador $<?$ es el operador primitivo para comparación de números naturales que decide si el primer operando es menor que el segundo, resulta en *tt* si así es, *ff* en otro caso. También, en la regla BEq, el operador $=?$ es el operador primitivo que decide si dos números naturales son iguales. Los escribimos así para distinguirlos de los operadores $<$ e $=$ que son símbolos del lenguaje.

2.4. Lógica de Hoare

Al igual que con los programas de While, se podrá especificar propiedades sobre los programas escritos en el lenguaje funcional simple mediante aserciones, predicados lógicos que hablarán sobre relaciones entre las variables del programa, que a diferencia de While tienen un valor fijo pues solo son un nombre para una constante, y el resultado.

Las aserciones definidas para el lenguaje While son predicados sobre los estados mientras que para el lenguaje funcional simple las aserciones tendrán que referirse al ambiente de variables para hablar del valor de una variable cuando sea necesario. Más aún, en el caso funcional también nos interesará hablar del resultado de la evaluación de un programa funcional que es un número natural, a diferencia de los programas While donde el resultado es un estado.

Con todo esto vemos que las ternas para el lenguaje funcional tendrán una definición un poco más compleja: las aserciones que funcionan como precondiciones ahora son predicados sobre ambientes de variables, y las poscondiciones son predicados sobre ambientes de variables y números naturales.

En Coq definimos los tipos para estas aserciones como sigue

Hoare.v
```
26 Definition rely := ambiente -> Prop.
27 Definition guarantee := nat -> ambiente -> Prop.
```

Aquí adoptamos como convención los nombres *rely* y *guarantee* que se usan en [13] para referirse a las aserciones con que se anotan programas escritos en el lenguaje PolyPCFv introducido en el mismo artículo, de manera análoga a las precondiciones y poscondiciones que se usaron para la lógica de While.

Estas definiciones funcionan tanto para programas como para expresiones aritméticas, pero para expresiones booleanas necesitaremos un tipo distinto para las poscondiciones dado que el resultado es un valor booleano

Hoare.v
```
28 Definition guarantee_bool := bool -> ambiente -> Prop.
```

Podemos pensar en el lenguaje de aserciones como un lenguaje de primer orden con igualdad tipificado, con dominio los números naturales, y operadores aritméticos, lógicos y desigualdad con su interpretación usual. Requerimos que sea tipificado pues tenemos distintos tipos de aserciones. Similarmente a la noción de satisfacibilidad para aserciones de la lógica de While discutida en la sección 1.3, decimos que un ambiente a *satisface* una aserción P de tipo `rely`, lo cual escribimos Pa, si y solo si $\mathcal{M} \models \mathcal{P}$, con $\mathcal{M}$ el modelo del lenguaje, y $\mathcal{P}$ se obtiene de reemplazar en P las variables libres con su respectivo valor según a. También, decimos que un ambiente a y número n *satisfacen* una aserción Q de tipo `guarantee`, lo cual escribimos Qna, si y solo si $\mathcal{M} \models \mathcal{Q}$, con $\mathcal{M}$ el modelo del lenguaje, y $\mathcal{Q}$ se obtiene de reemplazar en Q las variables libres con su respectivo valor según a (a n lo consideramos un término del dominio). Análogamente para aserciones de tipo `guarantee_bool`.

En la implementación no definimos un lenguaje para aserciones, estas son proposiciones en Coq (habitantes de `Prop`) donde cómodamente podemos expresar propiedades entre variables y valores del lenguaje funcional simple. Esto se puede apreciar claramente en las definiciones de los tipos `rely`, `guarantee` y `guarantee_bool`. Con esto encajamos las aserciones, y más adelante la lógica del lenguaje funcional simple, en la lógica de Coq como lo hacen los autores de [18], restringiendo cuantificadores a números naturales, valores booleanos y ambientes.

Al igual que cuando se definió la relación de evaluación, al definir las ternas de Hoare para los programas funcionales se asumirá que el ambiente de funciones está implícito y este contiene las definiciones necesarias para los nombres de funciones que son aplicadas en el texto de un programa.

Intuitivamente el significado de una terna de Hoare $\{P\}\ p\ \{Q\}$ es "si el programa p se ejecuta en un ambiente que satisface P, entonces el valor resultante satisface Q". Esto lo podemos escribir formalmente

$$\{P\}\ p\ \{Q\} \equiv \forall a\ n.\ p/a \Downarrow n \to Pa \to Qna \tag{2.3}$$

Nótese aquí que ambas aserciones P y Q hablan del mismo ambiente de variables a.

Análogamente, las definiciones de terna para expresiones aritméticas y booleanas se definen como

$$\{P\}\ e\ \{Q\} \equiv \forall a\ n.\ e/a \Downarrow n \to Pa \to Qna \tag{2.4}$$

$$\{P\}\ g\ \{Q\} \equiv \forall a\ b.\ g/a \Downarrow b \to Pa \to Qba \tag{2.5}$$

Algunos de los axiomas y reglas de inferencia para el lenguaje funcional simple tendrán cierto parecido con aquellas para programas imperativos; esto es de esperarse pues algunas estructuras de control tienen esencialmente el mismo comportamiento en un lenguaje imperativo como C que en uno funcional como Haskell, buen ejemplo son las expresiones condicionales if.

En contraste al caso imperativo, ahora podemos anotar la guardia de una expresión condicional, esto es deseable pues la guardia puede estar formada de expresiones aritméticas complicadas y/o programas por la manera en que se definió la gramática del lenguaje. Así, además de las ternas de Hoare que anotan cada rama del if (como se tiene en las hipótesis de la regla While-Hoare-Cond), añadimos una terna para anotar la guardia

$$\frac{\{P\}\ g\ \{Q'\} \qquad \{P \wedge Q'(tt)\}\ p_1\ \{Q\} \qquad \{P \wedge Q'(ff)\}\ p_2\ \{Q\}}{\{P\}\ \textbf{if}\ g\ \textbf{then}\ p_1\ \textbf{else}\ p_2\ \{Q\}}\ \textsc{HoareIf}$$

En esta regla se usan las propiedades de todas las subexpresiones de un programa, y no sólo de aquellas expresiones que pertenecen a la misma categoría sintáctica, para concluir una propiedad del programa mismo, demostrando una vez más la naturaleza composicional de la lógica de Hoare para el caso funcional así como se vio en el caso imperativo.

A diferencia de los programa escritos en While donde las expresiones aritméticas y booleanas sólo podían figurar en ciertos comandos, en este lenguaje funcional programas, expresiones aritméticas y expresiones booleanas están mutuamente definidas por lo que serán necesarias nuevas reglas para los operadores aritméticos y booleanos que no se tenían en la lógica de Hoare para While. Estas nuevas reglas también harán uso de las propiedades de las subexpresiones que existan para derivar ternas.

Las reglas para operadores aritméticos van a ser muy parecidas entre sí. Primero veamos el caso de la suma. Para poder concluir algo de una suma, nos apoyaremos de lo que podamos

concluir de sus operandos, de la misma manera que en el caso de la expresión condicional nos apoyamos de las ternas de las subexpresiones que la forman: si podemos concluir poscondiciones Q_1 para el primer operando y Q_2 para el segundo, requerimos que esta información sea suficiente para concluir una poscondición Q de la suma, es decir, debe ser el caso que

$$Q_1(n) \wedge Q_2(m) \to Q(n+m)$$

Aquí Q_1 habla del resultado n de evaluar el primer operando p_1, y Q_2 habla del resultado m de evaluar el segundo operando p_2. Entonces Q habla del resultado de la suma $n+m$. La regla para la suma es la siguiente

$$\frac{\{P\}\ p_1\ \{Q_1\} \qquad \{P\}\ p_2\ \{Q_2\} \qquad Q_1(n) \wedge Q_2(m) \to Q(n+m)}{\{P\}\ p_1 + p_2\ \{Q\}}\ \textsc{HoarePlus}$$

En general las reglas para operadores binarios tienen la misma forma. Si @ es un operador binario y tenemos como operandos expresiones e_1 y e_2, la regla para el operador tiene la forma

$$\frac{\{P\}\ e_1\ \{Q_1\} \qquad \{P\}\ e_2\ \{Q_2\} \qquad Q_1(c_1) \wedge Q_2(c_2) \to Q(c_1\ @\ c_2)}{\{P\}\ e_1\ @\ e_2\ \{Q\}}$$

donde c_1 y c_2 son los resultados de la evaluación de las expresiones e_1 y e_2 respectivamente. Tanto e_1 como e_2 se evalúan en el mismo ambiente por lo que la aserción P se mantiene como precondición para ambas. Las reglas para los operadores binarios del lenguaje se presentan en la figura 2.5c.

El caso de operadores con un solo operando es similar. Para cada operador unario # con operando una expresión e, podríamos proponer una regla de la forma

$$\frac{\{P\}\ e\ \{Q'\} \qquad Q'(c) \to Q(\#c)}{\{P\}\ \#e\ \{Q\}}$$

Pero al razonar sobre un programa y aplicar esta regla se requerirá realizar dos pruebas (resolver dos metas en Coq). En el caso de los operadores binarios no se puede evitar realizar tres pruebas pues los dos operandos son anotados con ternas y poseen dos poscondiciones distintas. Pero en este caso la regla se puede simplificar para requerir una sola prueba, una sola hipótesis en la regla

$$\frac{\{P\}\ e\ \{\lambda c.\ Q(\#c)\}}{\{P\}\ \#e\ \{Q\}}$$

Las reglas para operadores unarios del lenguaje se presentan en la figura 2.5d. En estas reglas se hace uso de la notación de función anónima (o expresión lambda) para escribir las

poscondiciones de las ternas de las hipótesis. Recordemos que las aserciones `rely`, `guarantee` y `guarantee_bool` en Coq se definieron como tipos de funciones, la notación de función anónima se usa para nombrar explícitamente el ambiente y/o el valor resultado de la evaluación del programa o expresión anotada, y que reciben las aserciones como argumento en las definiciones de terna (ecuaciones 2.3, 2.4 y 2.5).

La regla para una constante numérica en este lenguaje funcional simple tiene que tomar en cuenta el hecho de que una vez llegamos a una constante durante la ejecución de un programa ya no hay cómputos que realizar. A diferencia de otras construcciones sintácticas con subexpresiones donde hay cómputos inducidos por la evaluación de las subexpresiones y el procesamiento necesario de los resultados de estas para producir un valor, una constante no induce más cómputos pues ya es un valor.

La evaluación de una expresión aritmética que es simplemente una constante numérica suena sospechosamente parecida a la ejecución del comando **skip** de While que no produce ningún cómputo, entonces nos interesaría proponer un axioma para constantes con la forma del axioma While-Hoare-Skip que nos dice que la precondición es también la poscondición

$$\frac{}{\{Q\}\ n\ \{Q\}}$$

Sin embargo, la poscondición Q es un predicado sobre números naturales y ambientes de variables, y es obvio que el número del que habla Q en la poscondición es la constante n misma, entonces el axioma para constantes no puede tener simplemente Q como precondición y poscondición en la terna. No obstante, la idea detrás de este axioma va por buen camino y la solución a este pequeño detalle técnico es bastante sencilla: hacer que Q hable específicamente de n en la precondición.

Para justificar esto último revisitemos el axioma While-Hoare-Asig. Este nos dice que una aserción es cierta para la variable x (después de la asignación) si es cierta para el valor de la expresión a (antes de la asignación). De manera muy informal, este axioma lo único que hace es tomar una propiedad del sujeto (concretamente la expresión a, tal propiedad es la precondición escrita en la terna), ponerle un nuevo nombre "x" al sujeto y concluir la misma propiedad pues esto no afecta su validez.

Con esta idea en mente podemos razonar sobre una constante de la misma manera: podemos concluir que una aserción Q es cierta para la constante n si ya sabemos que es cierta para esa constante en particular, esto es, si sabemos $Q(n)$ de antemano. Entonces el axioma para constantes numéricas es

$$\frac{}{\{Q(n)\}\ n\ \{Q\}}\ \textsc{HoareNum}$$

Puesto de otra manera, este axioma nos dice que sólo es posible concluir propiedades de una constante que sean ciertas para esa constante. Más aún, el axioma es el mismo para constantes booleanas

$$\frac{}{\{Q(b)\}\ b\ \{Q\}}\ \textsc{HoareBool}$$

Y también para variables, pues como ya se mencionó, las variables son sólo nombres para referirnos a una constante numérica y esta asociación está dada por el ambiente

$$\frac{}{\{\lambda a.\ Q(a\ x)\}\ x\ \{Q\}}\ \textsc{HoareVar}$$

En el caso de declaraciones locales o expresiones let, la regla PLet nos dice explícitamente que ocurre un cambio en el ambiente durante la ejecución, este detalle deberá ser tratado cuidadosamente. Para una expresión **let** $x = p_1$ **in** p_2 primero se ejecuta la expresión a ligar p_1 en el ambiente de variables a para resolver el valor que va a ligar la variable x, seguido se ejecuta el cuerpo en el ambiente con la nueva definición del valor de x y que sólo existe dentro del cuerpo p_2.

Si P es la precondición de la expresión let, intuitivamente P también es precondición para p_1 pues el ambiente de variables no cambia y la evaluación de p_1 ocurre nada más alcanzar la expresión let durante la ejecución de un programa. Si podemos concluir una aserción R sobre p_1, dentro del cuerpo p_2 sabemos que R es válida para el valor de x, por lo que podríamos vernos tentados a usar la aserción R como precondición para p_2. También podríamos pensar que P funciona como precondición para p_2 pues la ejecución de p_1 no debería afectar su validez. Dado que el resultado de la evaluación del let es el resultado de la evaluación del cuerpo p_2, sería natural pensar que la poscondición para p_2 es también Q, dando lugar a la regla provisional

$$\frac{\{P\}\ p_1\ \{R\} \qquad \{P \wedge R(x)\}\ p_2\ \{Q\}}{\{P\}\ \textbf{let}\ x = p_1\ \textbf{in}\ p_2\ \{Q\}}$$

Sin embargo, esta regla es incorrecta porque no toma en cuenta el cambio de ambiente que ocurre antes de iniciar y tras finalizar la evaluación del cuerpo donde P y R no necesariamente se satisfacen. Para ejemplificar pensemos en una situación donde existen dos variables con el mismo nombre pero distinto valor, tal y como es el caso en el siguiente programa con dos declaraciones anidadas

$$\textbf{let}\ x = 2\ \textbf{in}\ (\textbf{let}\ x = 3\ \textbf{in}\ x)$$

Aquí identificamos dos variables: la "vieja" x con valor 2 (la más externa), y la "nueva" x con valor 3 (la más interna). Enfoquémonos en la declaración más interna, una precondición P para esta declaración se refiere a la vieja x, mientras que una precondición para su cuerpo se refiere

a la nueva x, y esta última puede no satisfacer las mismas propiedades que la primera, incluida P. Si $P \equiv Par$ y aplicamos la regla provisional

$$\frac{\{Par(x)\}\ 3\ \{R\} \qquad \{Par(x) \wedge R(x)\}\ x\ \{Q\}}{\{Par(x)\}\ \textbf{let}\ x = 3\ \textbf{in}\ x\ \{Q\}}$$

vemos que en el cuerpo de la declaración x vale 3, que no es par, y la precondición nos asegura que es el caso que $Par(x)$. Recordando la definición de terna (ecuación 2.3), tenemos un antecedente falso y podemos concluir cualquier poscondición Q, incluso aserciones falsas.

La regla como está escrita permite concluir cualquier poscondición para el cuerpo de una expresión let por vacuidad cada vez que la precondición de su cuerpo sea falsa, lo que puede volver inconsistente el sistema de inferencia. Como se dijo, el error está en no considerar el cambio de ambiente que ocurre al entrar al cuerpo de un let; en este ejemplo terminamos confundiendo una variable por otra.

La solución que proponemos sigue una idea similar a la regla de consecuencia While-Hoare-Cons, que es buscar adaptar la terna para el cuerpo con aserciones distintas S y T. Estas aserciones se refieren específicamente al nuevo ambiente de variables y nos permitirán concluir Q al salir del cuerpo. La regla propuesta es entonces

$$\frac{\begin{array}{c}\{P\}\ p_1\ \{R\} \qquad \{S\}\ p_2\ \{T\} \\ Pa \wedge Rna \to S(a[x \to n]) \\ Tm(a[x \to n]) \to Qma\end{array}}{\{P\}\ \textbf{let}\ x = p_1\ \textbf{in}\ p_2\ \{Q\}}\ \textsc{HoareLet}$$

Esta regla hace explícito el cambio en el ambiente de variables que ocurre durante la ejecución sin mencionar directamente a la semántica operacional, lo cual no sería deseable pues terminaríamos combinando la semántica operacional con la semántica axiomática que estamos definiendo con esta lógica estilo Hoare.

Cabe aclarar que esta regla surge de una necesidad práctica pues la definición de la semántica operacional de este lenguaje obedece las restricciones que se impusieron al principio de este capítulo, no obstante, la regla HoareLet se diseñó tratando de conservar la idea detrás de la regla provisional anterior. En la lógica propuesta en [13], la regla para expresiones let es muy parecida, por no decir idéntica, a la regla provisional, pero el lenguaje PolyPCFv y su lógica son diferentes a nuestro lenguaje funcional simple y no tienen que lidiar con la fuente del problema que son los ambientes de variables.

En el trabajo de Régis-Gianas y Pottier [18] se puede apreciar la misma idea: en su lenguaje la variable en una declaración local let se anota con una fórmula F, la regla de inferencia en la lógica propuesta requiere (1) probar que F es cierta para el valor de la expresión a ligar, y (2)

suponiendo F demostrar la poscondición para el cuerpo de la declaración, que es también la poscondición de la declaración misma. Está fórmula F cumple el mismo rol que la poscondición R en nuestra regla provisional.

Otra posible solución requeriría utilizar cuantificadores en las aserciones para recuperar, dentro del cuerpo, el ambiente de variables en el que se ejecuta la expresión let.

$$\frac{\{P\}\ p_1\ \{R\} \qquad \{\lambda a'.\ \exists a\ v.\ a' = a[x \to v] \wedge Pa \wedge Rva\}\ p_2\ \{\lambda v'\ a'.\ \forall a\ v.\ a' = a[x \to v] \to Qv'a\}}{\{P\}\ \textbf{let}\ x = p_1\ \textbf{in}\ p_2\ \{Q\}}$$

La idea detrás de esta regla, específicamente en la terna que involucra el cuerpo p_2, es recuperar el ambiente a en que se evalúa p_1 y el valor v resultante y hacer explícito el cambio de ambiente mediante la ecuación $a' = a[x \to v]$. La razón por la que preferimos la regla HoareLet es porque esta última es más complicada y su significado no es tan claro a primera vista.

Para poder dar una regla de inferencia para la aplicación de funciones es necesario hablar de las funciones y cómo deben ser tratadas en esta lógica, siguiendo la idea que sólo se necesita la especificación de una función para razonar con ella o con programas que la involucren.

Igual que con los programas, vamos a especificar el comportamiento de una función con aserciones, pero estas aserciones tienen un significado y propósito ligeramente distinto. La *precondición de una función* establece una restricción que debe cumplir el argumento que recibe, mientras que la *poscondición de una función* especifica una relación o propiedad del resultado que debe ser cierta asumiendo que el argumento satisface la precondición y que la evaluación de su cuerpo termina.

Dado que las funciones son cerradas, la precondición necesita referirse únicamente al argumento, y la poscondición necesita referirse al argumento y al resultado de la función (su cuerpo). Definimos entonces las precondiciones y poscondiciones para funciones como los tipos

Hoare.v
```
453 Definition pre := nat -> Prop.
454 Definition post := nat -> nat -> Prop.
```

Las funciones se anotan con una precondición Pre y una poscondición Post en una terna

$$\{\text{Pre}\}\ f\ x \Rightarrow p\ \{\text{Post}\} \equiv \forall n\ m.\ p/[x \to n] \Downarrow m \to \text{Pre}\ n \to \text{Post}\ nm \tag{2.6}$$

que debe ser adaptada a una terna sobre el cuerpo p, pues la especificación de la función debe poder demostrarse de su cuerpo ya que ese es el código de su implementación. Entonces

proponemos una nueva regla de inferencia para razonar con funciones

$$\frac{\{\lambda a.\ \text{Pre}\ (a\ x)\}\ p\ \{\lambda m\ a.\ \text{Post}\ (a\ x) m\}}{\{\text{Pre}\}\ f\ x \Rightarrow p\ \{\text{Post}\}}$$

Sin embargo esta regla no es suficiente para razonar con funciones recursivas como señaló Hoare en [12], donde extendió el lenguaje y las reglas del sistema de inferencia que propuso en su artículo original para considerar procedimientos recursivos en un lenguaje imperativo. Hoare nota que dentro de un procedimiento no es posible establecer propiedades para las invocaciones recursivas y en consecuencia no es posible demostrar ninguna propiedad del procedimiento. Efectivamente, cada vez que se requiera demostrar una propiedad para una invocación recursiva, esta a su vez requerirá demostrar propiedades para invocaciones recursivas, creando una cadena sin fin de propiedades por demostrar.

De la misma manera, no es posible demostrar propiedades del cuerpo de una función si esta contiene llamadas a sí misma, si su definición depende de sí misma. La solución que Hoare propone es simplemente asumir que las invocaciones recursivas de un procedimiento poseen la propiedad que se está tratando de demostrar. Esta solución se aplica también para funciones recursivas: asumimos la conclusión $\{\text{Pre}\}\ f\ x \Rightarrow p\ \{\text{Post}\}$ como hipótesis durante la prueba de la terna sobre el cuerpo p de f, lo que nos lleva a la regla más general

$$\frac{\begin{array}{c} cerrada\ f \\ \{\text{Pre}\}\ f\ x \Rightarrow p\ \{\text{Post}\} \vdash \{\lambda a.\ \text{Pre}\ (a\ x)\}\ p\ \{\lambda m\ a.\ \text{Post}\ (a\ x) m\} \end{array}}{\{\text{Pre}\}\ f\ x \Rightarrow p\ \{\text{Post}\}}\ \textsc{HoareFunc}$$

Esta regla nos permite asumir que toda llamada recursiva posee la propiedad que se está tratando de demostrar, pero no permite usar esta hipótesis para concluir trivialmente que la función posee tal propiedad. Más aún, si una función no es recursiva esa hipótesis nunca tendrá uso y esta regla cubre cómodamente tanto funciones recursivas como no recursivas. Adicionalmente pedimos que la función f sea cerrada para mantenernos fieles a la especificación del lenguaje.

Como nota al margen, podemos apreciar aquí un paralelo con las reglas de tipificado para el cálculo lambda con tipos simples extendido con un operador de recursión **fix**. La ya bien conocida regla de tipificado para este operador es

$$\frac{\Gamma, f : T \vdash e : T}{\Gamma \vdash \mathbf{fix}(f.e) : T}$$

donde f es la variable que representa la abstracción lambda recursiva cuyo tipo T se está tratando de derivar. Se asume que f tiene tipo T al tratar de derivar el tipo T del cuerpo e de f.

Sin embargo, existe una dificultad al tratar de implementar la regla HoareFunc en Coq. Informalmente las reglas de inferencia pueden verse como una forma de implicación: si se tienen las hipótesis entonces se tiene la conclusión; de manera similar con un juicio hipotético $A \vdash B$, se tiene el juicio B asumiendo el juicio A. Nuestro curso de acción sería implementar la regla HoareFunc como un teorema a demostrar con la forma $(A \to B) \to A$, tal y como se hizo con todas las demás reglas, donde A es la terna de Hoare para la función f y B la terna de Hoare para su cuerpo p. Y he aquí el problema: un teorema con tal estructura no puede demostrarse, en general, en el asistente de pruebas Coq.

A grandes rasgos, una demostración de un teorema con esta forma requiere demostrar A suponiendo $A \to B$, pero aún si A puede demostrarse a partir de B, es necesario tener una prueba de A para poder concluir B de la hipótesis $A \to B$ por *modus ponens*. En otras palabras, para demostrar A se necesita demostrar A. Evitamos este problema simplemente escribiendo la regla HoareFunc como un axioma en Coq.

Ahora que sabemos cómo tratar con las funciones del lenguaje podemos presentar la regla de inferencia para aplicación de funciones. Para razonar con funciones en una aplicación nos interesa su especificación (lo que la precondición y poscondición de la función nos aseguran de ella) y el argumento.

La poscondición con que se anota una función nos dice qué propiedades tiene el resultado siempre que el argumento satisface la precondición, por lo que si ya se ha demostrado que la función cumple con la especificación que fue proporcionada, resta demostrar que el argumento efectivamente satisface la precondición y que la poscondición de la función implica aquello que se quiera concluir de la aplicación. Entonces la regla para aplicación de funciones es la siguiente

$$\frac{\begin{array}{c}\{\text{Pre}\}\ f\ x \Rightarrow p\ \{\text{Post}\} \qquad \{P\}\ q\ \{R\} \\ Rna \to \text{Pre}\ n \\ Rna \wedge \text{Post}\ nm \to Qma\end{array}}{\{P\}\ f\ q\ \{Q\}}\ \textsc{HoareApp}$$

Al igual que en la lógica de Hoare definida para el lenguaje While, aquí damos la regla de consecuencia para poder adaptar las precondiciones y poscondiciones de una terna en un contexto de prueba dado a la forma de una terna en la conclusión de alguna regla del sistema de inferencia.

$$\frac{P \to P' \qquad \{P'\}\ p\ \{Q'\} \qquad Q' \to Q}{\{P\}\ p\ \{Q\}}\ \textsc{HoareConsecuencia}$$

La conclusión de esta regla es una terna para un programa, pero en este lenguaje funcional simple programas, expresiones aritméticas y expresiones booleanas pertenecen a categorías sintácticas distintas, además tenemos tres definiciones de terna en la implementación de la lógica en

Coq (ecuaciones 2.3, 2.4 y 2.5), por lo que en realidad tenemos tres reglas HoareConsecuencia, una por cada categoría sintáctica del lenguaje salvo funciones.

Los axiomas y reglas de inferencia que conforman la semántica axiomática estilo Hoare para el lenguaje funcional simple se compilan en la figura 2.5.

2.5. Implementación

La implementación en Coq del lenguaje funcional simple se puede consultar en línea, en el repositorio de código de este trabajo de tesis [14], se encuentra en los siguientes archivos del directorio `NFuncional`:

- `Gram.v` contiene la definición del lenguaje como se dio en la sección 2.2:
  - Se define la gramática del lenguaje (figura 2.1) con los tipos inductivos `programa` (programas P), `exp_arit` (expresiones aritméticas A) y `exp_bool` (expresiones booleanas B),
  - El tipo inductivo `funcion` para declaraciones de funciones,
  - Las funciones recursivas `fv`, `fv_arit` y `fv_bool`, mutuamente definidas, obtienen la lista de variables libres[4] en un programa, expresión aritmética y expresión booleana respectivamente,
  - El predicado `cerrada` (ecuación 2.2).
- `Eval.v` contiene la relación de evaluación para el lenguaje como se dio en la sección 2.3:
  - Se definen los tipos de dato `ambiente` para ambientes de variables y `ambiente_f` para ambientes de funciones,
  - Las relaciones de evaluación definidas inductivamente para programas `eval` (figura 2.2), expresiones aritméticas `eval_arit` (figura 2.3) y expresiones booleanas `eval_bool` (figura 2.4).
- `Hoare.v` contiene la lógica de Hoare presentada en la sección 2.4:
  - Se definen los tipos de dato `rely`, `guarantee` y `guarantee_bool`,
  - Las ternas de Hoare: `anotacion` para programas (ecuación 2.3), `anotacion_arit` para expresiones aritméticas (ecuación 2.4) y `anotacion_bool` para expresiones booleanas (ecuación 2.5),

[4] Representa el conjunto de variables libres $fv(\cdot)$.

$$\frac{\{P\}\ g\ \{Q'\} \qquad \{P \wedge Q'(tt)\}\ p_1\ \{Q\} \qquad \{P \wedge Q'(ff)\}\ p_2\ \{Q\}}{\{P\}\ \textbf{if}\ g\ \textbf{then}\ p_1\ \textbf{else}\ p_2\ \{Q\}}\ \textsc{HoareIf}$$

$$\frac{\begin{array}{c}\{P\}\ p_1\ \{R\} \qquad \{S\}\ p_2\ \{T\} \\ Pa \wedge Rna \rightarrow S(a[x \rightarrow n]) \\ Tm(a[x \rightarrow n]) \rightarrow Qma\end{array}}{\{P\}\ \textbf{let}\ x = p_1\ \textbf{in}\ p_2\ \{Q\}}\ \textsc{HoareLet}$$

$$\frac{\begin{array}{c}cerrada\ f \\ \{\text{Pre}\}\ f\ x \Rightarrow p\ \{\text{Post}\} \vdash \{\lambda a.\ \text{Pre}\ (a\ x)\}\ p\ \{\lambda m\ a.\ \text{Post}\ (a\ x) m\}\end{array}}{\{\text{Pre}\}\ f\ x \Rightarrow p\ \{\text{Post}\}}\ \textsc{HoareFunc}$$

$$\frac{\begin{array}{c}\{\text{Pre}\}\ f\ x \Rightarrow p\ \{\text{Post}\} \qquad \{P\}\ q\ \{R\} \\ Rna \rightarrow \text{Pre}\ n \\ Rna \wedge \text{Post}\ nm \rightarrow Qma\end{array}}{\{P\}\ f\ q\ \{Q\}}\ \textsc{HoareApp}$$

$$\frac{P \rightarrow P' \qquad \{P'\}\ p\ \{Q'\} \qquad Q' \rightarrow Q}{\{P\}\ p\ \{Q\}}\ \textsc{HoareConsecuencia}$$

(a) Reglas de inferencia para programas y funciones

$$\frac{}{\{Q(n)\}\ n\ \{Q\}}\ \textsc{HoareNum} \qquad \frac{}{\{Q(b)\}\ b\ \{Q\}}\ \textsc{HoareBool}$$

$$\frac{}{\{\lambda a.\ Q(a\ x)\}\ x\ \{Q\}}\ \textsc{HoareVar}$$

(b) Reglas de inferencia para constantes y variables

Figura 2.5: La lógica de Hoare para el lenguaje funcional simple

$$\frac{\{P\}\ p_1\ \{Q_1\} \qquad \{P\}\ p_2\ \{Q_2\} \qquad Q_1(n) \wedge Q_2(m) \rightarrow Q(n+m)}{\{P\}\ p_1 + p_2\ \{Q\}}\ \textsc{HoarePlus}$$

$$\frac{\{P\}\ p_1\ \{Q_1\} \qquad \{P\}\ p_2\ \{Q_2\} \qquad Q_1(n) \wedge Q_2(m) \rightarrow Q(n*m)}{\{P\}\ p_1 * p_2\ \{Q\}}\ \textsc{HoareProd}$$

$$\frac{\{P\}\ p_1\ \{Q_1\} \qquad \{P\}\ p_2\ \{Q_2\} \qquad Q_1(n) \wedge Q_2(m) \rightarrow Q(n =?\ m)}{\{P\}\ p_1 = p_2\ \{Q\}}\ \textsc{HoareEq}$$

$$\frac{\{P\}\ p_1\ \{Q_1\} \qquad \{P\}\ p_2\ \{Q_2\} \qquad Q_1(n) \wedge Q_2(m) \rightarrow Q(n <?\ m)}{\{P\}\ p_1 < p_2\ \{Q\}}\ \textsc{HoareLt}$$

$$\frac{\{P\}\ g_1\ \{Q_1\} \qquad \{P\}\ g_2\ \{Q_2\} \qquad Q_1(b_1) \wedge Q_2(b_2) \rightarrow Q(b_1\ \&\ b_2)}{\{P\}\ g_1\ \textbf{and}\ g_2\ \{Q\}}\ \textsc{HoareAnd}$$

(c) Reglas de inferencia para operadores binarios

$$\frac{\{P\}\ p\ \{\lambda n.\ Q(S\ n)\}}{\{P\}\ \textbf{succ}\ p\ \{Q\}}\ \textsc{HoareSucc} \qquad \frac{\{P\}\ p\ \{\lambda n.\ Q(pred\ n)\}}{\{P\}\ \textbf{pred}\ p\ \{Q\}}\ \textsc{HoarePred}$$

$$\frac{\{P\}\ g\ \{\lambda b.\ Q(\sim b)\}}{\{P\}\ \textbf{not}\ g\ \{Q\}}\ \textsc{HoareNot}$$

(d) Reglas de inferencia para operadores unarios

Figura 2.5: La lógica de Hoare para el lenguaje funcional simple (continuación)

- Precondiciones para funciones `pre`, poscondiciones para funciones `post` y ternas para funciones `anotacion_func`,
- Las reglas de inferencia de la lógica (figura 2.5), escritas como teoremas en Coq con su demostración, salvo la regla HoareFunc que fue declarada como axioma.

Los archivos `Ids.v`, `Maps.v` y `PMaps.v` son bibliotecas auxiliares con la implementación de identificadores para variables, funciones totales y funciones parciales, necesarias para la definición de los ambientes de variables y funciones de los que depende la implementación.

El archivo `Factorial.v` contiene un pequeño ejemplo de verificación formal de la función factorial en el lenguaje funcional simple usando la lógica desarrollada en este capítulo.

Salvo por las reglas HoareLet y HoareFunc, no hubo mayor complicación en la demostración de los axiomas y reglas de la lógica, de hecho varias demostraciones siguen en gran medida el mismo patrón, lo cual era de esperarse considerando que tenemos reglas con la misma estructura. Las reglas para operadores binarios (figura 2.5c) son un buen ejemplo, sus demostraciones siguen los mismos pasos:

1. Para una terna $\{P\}\ e_1@e_2\ \{Q\}$, asumimos las hipótesis de la regla para el operador @, la precondición P y que la ejecución termina $e_1@e_2/a \Downarrow v$. Por demostrar $Q(v)$.
2. Por inversión de la relación de evaluación sabemos que $v = c_1@c_2$, donde e_1 se evalúa a c_1 y e_2 se evalúa a c_2,
3. Por hipótesis sabemos que para demostrar $Q(c_1@c_2)$ basta demostrar $Q_1(c_1)$ y $Q_2(c_2)$,
4. Por definición de terna, para demostrar $Q_1(c_1)$ basta demostrar P y que $e_1/a \Downarrow c_1$, pero la primera submeta la sabemos por hipótesis (paso 1) y la segunda por inversión en el paso 2. Análogamente para $Q_2(c_2)$.

La situación es parecida con las demostraciones de las reglas de operadores unarios, y las reglas de constantes y variables, sus demostraciones siguen un cierto patrón y es evidente en el código. Por otro lado, la regla HoareFunc no tiene demostración pues es un axioma, mientras que la demostración de la regla HoareLet pudo realizarse una vez que se hizo explícito el cambio de ambiente de variables al entrar y salir del cuerpo de la expresión, como se discutió en su momento.

Recapitulando, en este capítulo definimos el lenguaje funcional simple, conformado por programas, expresiones aritméticas y booleanas, y declaraciones de funciones (figura 2.1), también definimos los ambientes de variables y de funciones, que son necesarios para definir la semántica operacional (figuras 2.2, 2.3 y 2.4). En la sección 2.4 presentamos las reglas de la

lógica de Hoare para el lenguaje funcional simple, primero definimos los tipos de las aserciones (`rely`, `guarantee` y `guarantee_bool`) y las anotaciones o ternas de Hoare (ecuaciones 2.3, 2.4 y 2.5) para su implementación, de igual manera con las *precondiciones*, *poscondiciones* y ternas para funciones (ecuación 2.6); con todo esto discutimos y diseñamos los axiomas y reglas de inferencia para las construcciones sintácticas del lenguaje (figura 2.5) y se implementaron en Coq (sección 2.5).

Capítulo 3

El lenguaje PCF

El lenguaje funcional simple presentado en el capítulo previo es muy reducido pero su simplicidad nos permitió explorar las ideas generales detrás de las estructuras de control funcionales y de cómo procede su evaluación para después plasmar estas nociones en reglas de inferencia, con particular interés en la definición y aplicación de funciones (tanto recursivas como no recursivas). En lenguajes de programación funcionales modernos las funciones son valores:[1] se pueden escribir funciones anónimas, una función puede tomar otra función como argumento o regresar una como resultado, etc., pero este comportamiento no fue permitido en nuestro lenguaje funcional simple.

En este capítulo trabajaremos con el lenguaje PCF[2] [9], un lenguaje que podemos considerar como una extensión del cálculo lambda con tipos simples $\lambda^{\rightarrow}$ con un operador primitivo de recursión comúnmente denotado en la literatura **fix**. Este operador **fix** simula el comportamiento de los operadores de punto fijo del cálculo lambda sin tipos, como el combinador Y, que permiten definir formalmente e implementar funciones recursivas.

En este capítulo nos ocuparemos de implementar en Coq una versión del lenguaje PCF utilizando la representación local anónima de variables, su relación de evaluación mediante una semántica operacional de paso grande, y su sistema de tipos, todo esto para tratar de implementar una lógica de Hoare como la que se definió en la sección 2.4, siguiendo la metodología descrita en [18] que consistirá en "encajar" el sistema de tipos de PCF y la lógica de Hoare propuesta en la lógica de Coq.

Para la implementación haremos uso de la biblioteca `TLC` [4], que contiene tipos de datos y operaciones que nos serán útiles para especificar relaciones acerca de las construcciones del lenguaje usando la representación local anónima.

[1] También se les suele llamar *objetos de primera clase*.

[2] Por su nombre en inglés *Programming language for Computable Functions*.

3.1. Representación local anónima

Para implementar el lenguaje PCF utilizaremos la representación local anónima como se presenta en [2]. Una descripción a mayor detalle del tema se encuentra en el apéndice A, en esta sección solo presentaremos las nociones básicas.

La representación local anónima se encarga de representar variables en la sintaxis de un lenguaje (el cálculo lambda, por ejemplo) combinando dos enfoques distintos: la representación con nombres y la representación anónima o con índices de De Bruijn.

En esta representación se hace una distinción sintáctica entre variables libres y variables ligadas. Las variables libres se representan con nombres o identificadores como x, y, etc., mientras que las variables ligadas se representan con índices que apuntan a la abstracción que liga a la variable; estos índices denotan el número de abstracciones (sus alcances) que existen entre la ocurrencia de la variable y su abstracción, informalmente, el número de lambdas que hay que "saltar" para encontrar la lambda que liga la variable.

Las construcciones sintácticas generadas por una gramática que emplea la representación local anónima se denominan *pretérminos*. Un *término localmente cerrado*, o simplemente *término*, es un pretérmino tal que toda variable ligada apunta a una abstracción presente en el término. Ejemplo: el pretérmino $\lambda.\lambda.0\ 1$ es un término localmente cerrado pues la variable ligada con índice 0 apunta a la abstracción más interna (no salta ninguna abstracción) y la variable ligada con índice 1 apunta a la abstracción más externa (salta una abstracción, la más interna), pero el pretérmino $\lambda.\lambda.0\ 2$ no es un término porque la variable ligada con índice 2 no apunta a ninguna de las dos abstracciones presentes.

Es importante asegurar que los pretérminos del lenguaje con los que se va a trabajar sean términos pues una variable ligada que no apunta a ninguna abstracción no tiene sentido en esta representación. En la representación anónima de variables, las variables libres son aquellas que apuntan a abstracciones inexistentes, pero en la representación local anónima estas ya son entes sintácticos distintos que no requieren índices.

Finalmente, una operación importante sobre los pretérminos es *abrir* el cuerpo de una abstracción con un término. Dada una abstracción $\lambda.t$ y un término u, el resultado de abrir la abstracción con u, denotado t^u, se obtiene al sustituir en t toda ocurrencia de la variable ligada por u. Esta operación es necesaria para determinar qué pretérminos son términos, y también para razonar con abstracciones o términos que ligan variables.

3.2. Definición del lenguaje PCF

La sintaxis de PCF está dada por las reglas en notación estilo BNF en la figura 3.1. Los valores pertenecen a su propia categoría sintáctica, distinta a los pretérminos del lenguaje, esto será útil al definir la relación de evaluación de PCF.

			Pretérminos
t	$::=$	i	*Variable ligada*
	$\mid$	x	*Variable libre*
	$\mid$	v	*Valor*
	$\mid$	$t_1\ t_2$	*Aplicación*
	$\mid$	$t_1 + t_2$	*Suma*
	$\mid$	$t_1 * t_2$	*Producto*
	$\mid$	**succ** t	*Sucesor*
	$\mid$	**pred** t	*Predecesor*
	$\mid$	**iszero** t	*Test cero*
	$\mid$	**if** t_1 **then** t_2 **else** t_3	*Condicional*
			Valores
v	$::=$	n	*Número natural*
	$\mid$	*tt*	*true*
	$\mid$	*ff*	*false*
	$\mid$	$\lambda_{(U,T)(P,Q)}.t$	*Abstracción lambda*
	$\mid$	$\mu_{(U,T)(P,Q)}.t$	*Abstracción fix*

Figura 3.1: El lenguaje PCF (sintaxis)

El lenguaje PCF tiene como tipos básicos los números naturales y las constantes booleanas, y cuenta con operaciones básicas primitivas para sus habitantes. El lenguaje cuenta también con tipos función y admite funciones anónimas y recursivas. Los valores de PCF se conforman de números naturales, constantes booleanas y funciones.

Al emplear la representación local anónima de variables, hacemos una distinción sintáctica entre variables libres y variables ligadas. Los índices de variables ligadas se implementan con números naturales.

Las estructuras de control de este lenguaje son las expresiones condicionales y la aplicación de función. En este lenguaje no tenemos expresiones let como en el lenguaje funcional simple pues en esta implementación de PCF las variables ligadas no tienen nombre, así no tiene sentido

tener expresiones let cuyo propósito es ligar localmente una expresión a un nombre.

En esta implementación de PCF introducimos un tipo especial de abstracción que servirá para definir funciones recursivas, buscando simular el comportamiento del operador **fix** previamente mencionado. Esta nueva abstracción fix, escrita usando la letra griega μ, puede verse como el operador **fix** seguido de una abstracción lambda

$$\mu f x.t := \mathbf{fix}(f.(\lambda x.t)) \tag{3.1}$$

pero usando índices en vez de nombres para las variables ligadas. La adición de esta nueva abstracción en vez del operador **fix** es por mera comodidad, haciendo claro el hecho de que una abstracción fix es una función.

Al igual que las abstracciones lambda, las abstracciones fix ya son valores y no inducen cómputos salvo que estén presentes en el lado izquierdo de una aplicación. Con esto evitamos ciclos infinitos que pueden aparecer ingenuamente, ejemplo clásico es la expresión $\mathbf{fix}(x.x)$ que se reduce a sí misma. Una abstracción lambda sólo liga una variable dentro de su cuerpo t con el índice 0, esta es su parámetro; en una abstracción fix se ligan dos variables, el parámetro de la función con el índice 0 (variable x en 3.1) y la función misma con el índice 1 (variable f en 3.1).

En la definición podemos ver que las abstracciones están explícitamente anotadas con el tipo U de su parámetro y el tipo T de regreso (lo que facilitará la definición de la relación de tipificado), junto con un par de fórmulas P y Q que serán de utilidad más adelante[3].

Como se había mencionado en la sección anterior, es necesario seleccionar los pretérminos de la gramática que serán los términos del lenguaje. En la figura 3.2 se define el predicado $term$ para seleccionar los términos de PCF. La idea general es simple: un término es una variable libre, una constante, o un pretérmino tal que toda subexpresión es un término. Analizamos con más detenimiento las abstracciones pues estas introducen variables ligadas en el lenguaje.

Para determinar que una abstracción $\lambda.t$ sea un término debemos analizar su cuerpo t, pero no es posible simplemente extraer el cuerpo de la expresión pues esto deja toda instancia de la variable ligada apuntando a una abstracción inexistente. Primero se debe reemplazar toda instancia de la variable ligada al pasar de la abstracción, para ello abrimos el cuerpo t con una variable libre x fresca[4], y si la expresión t^x es un término, entonces la abstracción $\lambda.t$ es un término. En el caso de una abstracción $\mu.t$ abrimos el cuerpo t con dos variables libres distintas, f y x, frescas, y si $t^{f,x}$ es un término, $\mu.t$ es un término.

[3]Por legibilidad obviaremos estas anotaciones al hablar sobre abstracciones en el presente texto, pero no en definiciones o figuras que las involucren.

[4]En la literatura suele ser norma el escoger esta variable x fresca como una nueva variable distinta de todas las variables libres ya presentes en el cuerpo de la abstracción, es decir, $x \notin fv(t)$. La representación local anónima presentada en [2] es más general en este aspecto, solo requiere que exista un conjunto finito de nombres de variables L tal que $x \notin L$; este conjunto puede ser $fv(t)$. Ver apéndice A.

$$\frac{}{term\ x} \qquad \frac{}{term\ n} \qquad \frac{}{term\ \mathit{tt}} \qquad \frac{}{term\ \mathit{ff}}$$

$$\frac{\forall x \notin L.\quad term\ t^x}{term\ \lambda_{(U,T)(P,Q)}.t} \qquad \frac{\forall f \notin L. \forall x \notin L \cup \{f\}.\quad term\ t^{f,x}}{term\ \mu_{(U,T)(P,Q)}.t}$$

$$\frac{term\ t_1 \qquad term\ t_2}{term\ t_1\ t_2} \qquad \frac{term\ t_1 \qquad term\ t_2}{term\ t_1 + t_2} \qquad \frac{term\ t_1 \qquad term\ t_2}{term\ t_1 * t_2} \qquad \frac{term\ t}{term\ \mathbf{succ}\ t}$$

$$\frac{term\ t}{term\ \mathbf{pred}\ t} \qquad \frac{term\ t}{term\ \mathbf{iszero}\ t} \qquad \frac{term\ t_1 \qquad term\ t_2 \qquad term\ t_3}{term\ \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3}$$

Figura 3.2: Términos de PCF

La operación *abrir* se define recursivamente sobre la estructura de los pretérminos. Definimos una función auxiliar $\{k \rightsquigarrow u\}\ t$ que reemplaza en t toda instancia de la variable ligada con índice k por el término u

$$\{k \rightsquigarrow u\}\ i = \begin{cases} u & \text{si } i = k \\ i & \text{e.o.c.} \end{cases}$$

y recorre el índice de la variable ligada que se quiere reemplazar cada vez que pasamos por una abstracción lambda o fix

$$\{k \rightsquigarrow u\}(\lambda_{(U,T)(P,Q)}.t) = \lambda_{(U,T)(P,Q)}.\{k+1 \rightsquigarrow u\}\ t$$

$$\{k \rightsquigarrow u\}(\mu_{(U,T)(P,Q)}.t) = \mu_{(U,T)(P,Q)}.\{k+2 \rightsquigarrow u\}\ t$$

Dado que la variable que se está ligando en una abstracción $\lambda.t$ tiene índice 0 en el cuerpo, la operación abrir se define como

$$t^u := \{0 \rightsquigarrow u\}\ t \tag{3.2}$$

En el caso de una abstracción $\mu.t$ donde se ligan dos variables, abrimos la abstracción con dos términos s, u simultáneamente

$$t^{s,u} := \{0 \rightsquigarrow u\}\ \{1 \rightsquigarrow s\}\ t \tag{3.3}$$

La implementación del lenguaje PCF en Coq se encuentra en el archivo `PCF_Lang.v`. La definición del lenguaje se implementa como dos tipos inductivos mutuamente definidos para pretérminos y valores como aparecen en la figura 3.1.

PCF_Lang.v

```
Inductive trm : Type :=
| trm_bvar : nat -> trm
| trm_fvar : var -> trm
| trm_val  : val -> trm
| trm_app  : trm -> trm -> trm
| trm_plus : trm -> trm -> trm
| trm_prod : trm -> trm -> trm
| trm_suc  : trm -> trm
| trm_pred : trm -> trm
| trm_iz   : trm -> trm
| trm_if   : trm -> trm -> trm -> trm
with val : Type :=
     | trm_num  : nat -> val
     | trm_tt   : val
     | trm_ff   : val
     | trm_abs  : forall (U T : typ), ([U] -> Prop) -> ([U] -> [T] -> Prop)
         -> trm -> val
     | trm_fix  : forall (U T : typ), ([U] -> Prop) -> ([U] -> [T] -> Prop)
         -> trm -> val.
```

También se define la operación abrir, primero definiendo la función auxiliar $\{k \rightsquigarrow u\}\ t$, junto con la notación usada para hacer el código más legible

PCF_Lang.v

```
Reserved Notation "{ k '~>' u } t" (at level 67, right associativity).

(* Función recursiva para abrir una variable ligada `k` dentro un término `t`
 * con otro término `u`. *)
Fixpoint open_rec (k : nat) (u : trm) (t : trm) {struct t} : trm :=
  match t with
  | trm_bvar i     => if k =? i then u else t
  | trm_fvar _     => t
  | trm_val v      => trm_val (open_rec_val k u v)
  | trm_app t1 t2  => trm_app ({ k ~> u } t1) ({ k ~> u } t2)
  | trm_plus t1 t2 => trm_plus ({ k ~> u } t1) ({ k ~> u } t2)
  | trm_prod t1 t2 => trm_prod ({ k ~> u } t1) ({ k ~> u } t2)
  | trm_suc t1     => trm_suc ({ k ~> u } t1)
```

```
68    | trm_pred t1    => trm_pred ({ k ~> u } t1)
69    | trm_iz t1      => trm_iz ({ k ~> u } t1)
70    | trm_if t1 t2 t3 => trm_if ({ k ~> u } t1) ({ k ~> u } t2) ({ k ~> u }
          t3)
71    end
72  with open_rec_val (k : nat) (u : trm) (v : val) {struct v} : val :=
73         match v with
74         | trm_abs U T P Q t1  => trm_abs U T P Q ({ S k ~> u } t1)
75         | trm_fix U T P Q t1  => trm_fix U T P Q ({ S (S k) ~> u } t1)
76         | _                  => v
77         end
78
79  where "{ k ~> u } t" := (open_rec k u t).
```

entonces podemos definir la operación abrir como se ve en las ecuaciones 3.2 y 3.3

PCF_Lang.v

```
81  Definition open t u := {0 ~> u} t.
82  Definition open_t t s u := {0 ~> u} {1 ~> s} t.
83
84  Notation "t ^^ u" := (open t u) (at level 67).
```

Por último, definimos el predicado $term$

PCF_Lang.v

```
88  Inductive term : trm -> Prop :=
89  | term_var : forall x,
90      term (trm_fvar x)
91  | term_v : forall v,
92      term_val v -> term v
93  | term_app : forall t1 t2,
94      term t1 ->
95      term t2 ->
96      term (trm_app t1 t2)
97  | term_plus : forall t1 t2,
98      term t1 ->
99      term t2 ->
100     term (trm_plus t1 t2)
101 | term_prod : forall t1 t2,
```

```
102      term t1 ->
103      term t2 ->
104      term (trm_prod t1 t2)
105  | term_suc : forall t1,
106      term t1 ->
107      term (trm_suc t1)
108  | term_pred : forall t1,
109      term t1 ->
110      term (trm_pred t1)
111  | term_iz : forall t1,
112      term t1 ->
113      term (trm_iz t1)
114  | term_if : forall t1 t2 t3,
115      term t1 ->
116      term t2 ->
117      term t3 ->
118      term (trm_if t1 t2 t3)
119  with term_val : val -> Prop :=
120       | term_num : forall n,
121           term_val (trm_num n)
122       | term_tt :
123           term_val tt
124       | term_ff :
125           term_val ff
126       | term_abs : forall L U T P Q t1,
127           (forall x, x \notin L -> term (t1 ^^ trm_fvar x)) ->
128           term_val (trm_abs U T P Q t1)
129       | term_fix : forall L U T P Q t1,
130           (forall f, f \notin L ->
131            forall x, x \notin L \u \{f} ->
132                      term (open_t t1 (trm_fvar f) (trm_fvar x))) ->
133           term_val (trm_fix U T P Q t1).
```

Dado que pretérminos y valores de PCF son categorías sintácticas distintas en la implementación, cuyas definiciones dependen una de la otra, las relaciones y funciones que se definan sobre los elementos del lenguaje requerirán a su vez ser implementadas en pares, una definición sobre pretérminos `trm` y otra sobre valores `val`.

3.3. Semántica operacional de paso grande para PCF

La evaluación de programas en PCF está dada por una semántica operacional de paso grande, su definición requiere de un ambiente de variables E y la estrategia de evaluación es llamada por valor. El ambiente es una colección finita de pares (x, v) que asocia una variable libre x con un valor v, lo cual escribimos $E(x) = v$. El ambiente E tiene la restricción que cada variable tiene a lo más un valor asociado a ella.

La relación

$$E \Vdash t \Downarrow v$$

se lee "el término t en el ambiente E se evalúa al valor v". Las reglas de evaluación para términos se presentan en la figura 3.3.

$$\frac{E(x) = v \qquad term\ v}{E \Vdash x \Downarrow v} \qquad \frac{term\ v}{E \Vdash v \Downarrow v}$$

$$\frac{E \Vdash t_1 \Downarrow \lambda_{(U,T)(P,Q)}.t_3 \quad E \Vdash t_2 \Downarrow v_2 \quad E \Vdash {t_3}^{v_2} \Downarrow v_3}{E \Vdash t_1\ t_2 \Downarrow v_3} \qquad \frac{E \Vdash t_1 \Downarrow \mu_{(U,T)(P,Q)}.t_3 \quad E \Vdash t_2 \Downarrow v_2 \quad E \Vdash {t_3}^{\mu,v_2} \Downarrow v_3}{E \Vdash t_1\ t_2 \Downarrow v_3}$$

$$\frac{E \Vdash t_1 \Downarrow n \qquad E \Vdash t_2 \Downarrow m}{E \Vdash t_1 + t_2 \Downarrow n + m} \qquad \frac{E \Vdash t_1 \Downarrow n \qquad E \Vdash t_2 \Downarrow m}{E \Vdash t_1 * t_2 \Downarrow n * m}$$

$$\frac{E \Vdash t \Downarrow n}{E \Vdash \mathbf{succ}\ t \Downarrow S\ n} \qquad \frac{E \Vdash t \Downarrow n}{E \Vdash \mathbf{pred}\ t \Downarrow pred\ n}$$

$$\frac{E \Vdash t \Downarrow 0}{E \Vdash \mathbf{iszero}\ t \Downarrow \mathit{tt}} \qquad \frac{E \Vdash t \Downarrow S\ n}{E \Vdash \mathbf{iszero}\ t \Downarrow \mathit{ff}}$$

$$\frac{E \Vdash t_1 \Downarrow \mathit{tt} \qquad E \Vdash t_2 \Downarrow v}{E \Vdash \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 \Downarrow v} \qquad \frac{E \Vdash t_1 \Downarrow \mathit{ff} \qquad E \Vdash t_3 \Downarrow v}{E \Vdash \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 \Downarrow v}$$

Figura 3.3: Relación de evaluación para PCF

Hacemos uso de la operación abrir en la aplicación de funciones como un análogo de la β-reducción (en una semántica operacional de paso pequeño) para la representación local anónima: una aplicación $(\lambda.t)v$ se "reduce" a t^v. De igual manera, una aplicación $(\mu.t)v$ se "reduce" a $t^{\mu,v}$,

donde el superíndice μ es abreviación para la abstracción fix $\mu.t$, esto es, el cuerpo de la función se abre con sí misma y con el argumento, en un espíritu similar a la reducción del operador $\mathbf{fix}(f.e)$ para "desenvolverlo" en $e[\mathbf{fix}(f.e)/f]$.

Para ver con más claridad de dónde proviene la regla para aplicación de abstracciones fix en la figura 3.3, tomemos las reglas de evaluación de paso grande y llamada por valor para aplicación de funciones y el operador **fix**[5]

$$\frac{t_2 \longrightarrow v \qquad t_1 \longrightarrow \lambda x.t \qquad t[v/x] \longrightarrow w}{t_1\ t_2 \longrightarrow w} \qquad \frac{e[\mathbf{fix}(f.e)/f] \longrightarrow v}{\mathbf{fix}(f.e) \longrightarrow v}$$

y recordemos la ecuación 3.1, que nos dice que podemos ver una abstracción fix $\mu fx.t$ como **fix** seguido de una abstracción lambda, ahora pensemos en la siguiente derivación de una aplicación de una abstracción fix

$$\frac{\dfrac{\vdots}{t_2 \longrightarrow v} \qquad \dfrac{\dfrac{\vdots}{(\lambda x.t)[\mu fx.t/f] \longrightarrow \lambda x.t'}}{t_1 = \mu fx.t \longrightarrow \lambda x.t'} \qquad \dfrac{\vdots}{t'[v/x] \longrightarrow w}}{t_1\ t_2 \longrightarrow w}$$

Supongamos por un momento que la sustitución $(\lambda x.t)[\mu fx.t/f]$ está bien definida, entonces tenemos que $t' = t[\mu fx.t/f]$, y reescribimos la derivación anterior como

$$\frac{\dfrac{\vdots}{t_2 \longrightarrow v} \qquad \dfrac{\dfrac{(\lambda x.t)[\mu fx.t/f] = \lambda x.t[\mu fx.t/f]}{(\lambda x.t)[\mu fx.t/f] \longrightarrow \lambda x.t[\mu fx.t/f]}}{t_1 = \mu fx.t \longrightarrow \lambda x.t[\mu fx.t/f]} \qquad \dfrac{\vdots}{t[\mu fx.t/f][v/x] \longrightarrow w}}{t_1\ t_2 \longrightarrow w}$$

En nuestra presentación de PCF, $(\mu.t)v$ se "reduce" o evalúa a $t^{\mu,v}$ de la misma forma que $t_1\ t_2 = (\mu fx.t)t_2$ se reduce a $t[\mu fx.t/f][v/x]$ en la derivación anterior.

Aunque se puede definir la relación de evaluación sin un ambiente de variables tal que la evaluación de un término no cerrado tarde o temprano se bloquea al alcanzar una variable libre, como suele ser norma, este tendrá una función más adelante cuando definamos la semántica axiomática de PCF. La relación de evaluación como está definida aún se puede bloquear si una variable no está definida en el ambiente durante la ejecución.

La implementación de la relación de evaluación `reds` se encuentra en `PCF_Eval.v`. El ambiente de variables se define como un ambiente de valores usando el tipo de dato `env` de la

[5]En el libro de Dowek [9] se puede encontrar una presentación tradicional del lenguaje PCF como el cálculo lambda extendido con **fix**, su semántica operacional de paso grande, así como una descripción de distintas estrategias de evaluación.

biblioteca `TLC`. Dado que un valor no induce cómputos, la relación de evaluación se define sobre términos `trm`, y el resultado de la evaluación de un término debe ser un valor `val`. También se proporciona notación.

PCF_Eval.v

```
(* Ambientes de evaluación con las definiciones de variables libres. *)
Definition eval_env := env val.

Reserved Notation "E ||- t ⇓ v" (at level 68).

Inductive reds : eval_env -> trm -> val -> Prop :=
(* Variables libres. *)
| reds_var : forall E x v,
    ok E ->
    binds x v E ->
    term_val v ->
    E ||- trm_fvar x ⇓ v
(* Valores. *)
| reds_val : forall E v,
    term_val v ->
    E ||- trm_val v ⇓ v
(* Apliación de una abstracción. *)
| reds_red : forall E U T P Q t3 v2 v3 t1 t2,
    E ||- t1 ⇓ (trm_abs U T P Q t3) ->
    E ||- t2 ⇓ v2 ->
    E ||- t3 ^^ v2 ⇓ v3 ->
    E ||- trm_app t1 t2 ⇓ v3
(* Apliación de una función recursiva. *)
| reds_fix : forall E U T P Q t3 v2 v3 t1 t2,
    let fix_ := trm_fix U T P Q t3 in
    E ||- t1 ⇓ fix_ ->
    E ||- t2 ⇓ v2 ->
    E ||- open_t t3 fix_ v2 ⇓ v3 ->
    E ||- trm_app t1 t2 ⇓ v3
(* Suma. *)
| reds_plus : forall E n m t1 t2,
    E ||- t1 ⇓ trm_num n ->
```

```
47      E ||- t2 ⇓ trm_num m ->
48      E ||- trm_plus t1 t2 ⇓ trm_num (n + m)
49  (* Producto. *)
50  | reds_prod : forall E n m t1 t2,
51      E ||- t1 ⇓ trm_num n ->
52      E ||- t2 ⇓ trm_num m ->
53      E ||- trm_prod t1 t2 ⇓ trm_num (n * m)
54  (* Sucesor. *)
55  | reds_suc : forall E n t1,
56      E ||- t1 ⇓ trm_num n ->
57      E ||- trm_suc t1 ⇓ trm_num (S n)
58  (* Predecesor. *)
59  | reds_pred : forall E n t1,
60      E ||- t1 ⇓ trm_num n ->
61      E ||- trm_pred t1 ⇓ trm_num (pred n)
62  (* IsZero, caso verdadero. *)
63  | reds_iz_tt : forall E t1,
64      E ||- t1 ⇓ trm_num 0 ->
65      E ||- trm_iz t1 ⇓ tt
66  (* IsZero, caso falso. *)
67  | reds_iz_ff : forall E n t1,
68      E ||- t1 ⇓ trm_num (S n) ->
69      E ||- trm_iz t1 ⇓ ff
70  (* If, guardia verdadera. *)
71  | reds_if_tt : forall E v g t1 t2,
72      E ||- g ⇓ tt ->
73      E ||- t1 ⇓ v ->
74      E ||- trm_if g t1 t2 ⇓ v
75  (* If, guardia falsa. *)
76  | reds_if_ff : forall E v g t1 t2,
77      E ||- g ⇓ ff ->
78      E ||- t2 ⇓ v ->
79      E ||- trm_if g t1 t2 ⇓ v
80
81  where "E ||- t ⇓ v" := (reds E t v).
```

3.4. El sistema de tipos para PCF

Los tipos de PCF se presenta en la figura 3.4. La relación o juicio de tipificado[6]

$$\Gamma \vDash t : T$$

se lee "el término t tiene tipo T bajo el contexto Γ" y está definida en la figura 3.5[7]. El contexto de tipos (*typing context*) Γ es una colección finita de pares (x, T) que asocia una variable libre x con un tipo T, que escribimos $x : T$. El contexto Γ tiene la restricción que, dada una variable x, hay a lo más un tipo T tal que $x : T \in \Gamma$.

			Tipos
T	$::=$	N	*Números naturales*
	$\mid$	B	*Constantes booleanas*
	$\mid$	$T_1 \Longrightarrow T_2$	*Funciones*

Figura 3.4: Los tipos de PCF

$$\frac{x : T \in \Gamma}{\Gamma \vDash x : T} \qquad \frac{}{\Gamma \vDash n : N} \qquad \frac{}{\Gamma \vDash \mathit{tt} : B} \qquad \frac{}{\Gamma \vDash \mathit{ff} : B}$$

$$\frac{\forall x \notin L. \quad \Gamma, x : U \vDash t^x : T}{\Gamma \vDash \lambda_{(U,T)(P,Q)}.t : U \Longrightarrow T} \qquad \frac{\forall f \notin L. \forall x \notin L \cup \{f\}. \quad \Gamma, f : U \Longrightarrow T, x : U \vDash t^{f,x} : T}{\Gamma \vDash \mu_{(U,T)(P,Q)}.t : U \Longrightarrow T}$$

$$\frac{\Gamma \vDash t_1 : U \Longrightarrow T \qquad \Gamma \vDash t_2 : U}{\Gamma \vDash t_1\ t_2 : T} \qquad \frac{\Gamma \vDash t_1 : N \qquad \Gamma \vDash t_2 : N}{\Gamma \vDash t_1 + t_2 : N}$$

$$\frac{\Gamma \vDash t_1 : N \qquad \Gamma \vDash t_2 : N}{\Gamma \vDash t_1 * t_2 : N} \qquad \frac{\Gamma \vDash t : N}{\Gamma \vDash \mathbf{succ}\ t : N} \qquad \frac{\Gamma \vDash t : N}{\Gamma \vDash \mathbf{pred}\ t : N} \qquad \frac{\Gamma \vDash t : N}{\Gamma \vDash \mathbf{iszero}\ t : B}$$

$$\frac{\Gamma \vDash t_1 : B \qquad \Gamma \vDash t_2 : T \qquad \Gamma \vDash t_3 : T}{\Gamma \vDash \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 : T}$$

Figura 3.5: Juicios de tipificado para PCF

[6]También se le suele llamar relación para verificación de tipos.

[7]El uso del doble torniquete $\vDash$, en vez del torniquete $\vdash$ como es usual en la literatura, es solo para notación y no debe entenderse como una forma de consecuencia semántica. El uso del torniquete $\vdash$ lo reservamos como notación para la semántica axiomática más adelante.

La implementación de los tipos `typ` se encuentra en `PCF_Types.v`, cuya definición refleja directamente la estructura de la figura 3.4.

PCF_Types.v

```
Inductive typ : Set :=
| typ_nat   : typ
| typ_bool  : typ
| typ_arrow : typ -> typ -> typ.

(* Notaciones. *)
Notation "'N'" := (typ_nat).
Notation "'B'" := (typ_bool).
Notation "U '==>' T" := (typ_arrow U T) (at level 2, right associativity).
```

La relación de tipificado se encuentra en `PCF_Typing.v`. Definimos el contexto de tipos `ctx` como un ambiente `env` de tipos `typ`, las relaciones de tipificado sobre términos y valores, `typing` y `typing_val` respectivamente, y se proporciona notación.

PCF_Typing.v

```
(* Contextos de tipos. *)
Definition ctx := env typ.

(* Relación de tipificado. *)
Reserved Notation "G |= t ~: T" (at level 69).
Reserved Notation "G |= v ≈: T" (at level 69).

Inductive typing : ctx -> trm -> typ -> Type :=
| typing_var : forall G x T,
    ok G ->
    binds x T G ->
    G |= trm_fvar x ~: T
| typing_v : forall G v T,
    G |= v ≈: T ->
    G |= v ~: T
| typing_app : forall U T G t1 t2,
    G |= t1 ~: U ==> T ->
    G |= t2 ~: U ->
    G |= trm_app t1 t2 ~: T
| typing_plus : forall G t1 t2,
```

```
26      G |= t1 ~: N ->
27      G |= t2 ~: N ->
28      G |= trm_plus t1 t2 ~: N
29  | typing_prod : forall G t1 t2,
30      G |= t1 ~: N ->
31      G |= t2 ~: N ->
32      G |= trm_prod t1 t2 ~: N
33  | typing_suc : forall G t1,
34      G |= t1 ~: N ->
35      G |= trm_suc t1 ~: N
36  | typing_pred : forall G t1,
37      G |= t1 ~: N ->
38      G |= trm_pred t1 ~: N
39  | typing_iz : forall G t1,
40      G |= t1 ~: N ->
41      G |= trm_iz t1 ~: B
42  | typing_if : forall T G t1 t2 t3,
43      G |= t1 ~: B ->
44      G |= t2 ~: T ->
45      G |= t3 ~: T ->
46      G |= trm_if t1 t2 t3 ~: T
47
48  where "G |= t ~: T" := (typing G t T)
49
50  with typing_val : ctx -> val -> typ -> Type :=
51       | typing_num : forall G n,
52           G |= trm_num n ≈: N
53       | typing_tt : forall G,
54           G |= tt ≈: B
55       | typing_ff : forall G,
56           G |= ff ≈: B
57       | typing_abs : forall L G U T P Q t1,
58           (forall x, x \notin L -> G & x ~ U |= t1 ^^ trm_fvar x ~: T) ->
59           G |= trm_abs U T P Q t1 ≈: U ==> T
60       | typing_fix : forall L G U T P Q t1,
61           (forall f, f \notin L ->
```

```
62            forall x, x \notin L \u \{f} ->
63                      G & f ~ U ==> T & x ~ U |= open_t t1 (trm_fvar f)
                            (trm_fvar x) ~: T) ->
64          G |= trm_fix U T P Q t1 ≈: U ==> T
65
66  where "G |= v ≈: T" := (typing_val G v T).
```

3.5. Lógica de Hoare para PCF

Antes de poder definir las aserciones e implementar el sistema de inferencia vamos a encajar, o reflejar, los tipos y valores de PCF en la lógica de Coq. Cuando hablemos de tipos o valores de PCF diremos que pertenecen al *nivel computacional*, cuando hablemos de términos de la lógica de Coq diremos que pertenecen al *nivel lógico*.

A cada tipo en el nivel computacional le asignaremos un tipo en el nivel lógico (esto es, en `Type`), y a cada valor del nivel computacional un término del nivel lógico que habita en la reflexión lógica del tipo computacional del valor. Como es de esperarse, la reflexión lógica de los números naturales en PCF son los números naturales de Coq, análogamente para las constantes booleanas. Para las funciones adoptamos el método presentado en [18].

Recordemos que las funciones de PCF se anotan explícitamente no solo con los tipos del parámetro y de retorno, también con un par de fórmulas P y Q (figura 3.1); estas fórmulas fungen como la precondición y la poscondición de una función respectivamente, en el mismo espíritu que las aserciones Pre y Post con que se anota una función del lenguaje funcional simple en una terna (ecuación 2.6). Estas fórmulas, al igual que las aserciones con que anotaremos programas, pertenecen a una lógica de orden superior pues, de acuerdo a los autores de [18], si las funciones pueden abstraer sobre funciones, entonces especificaciones deben abstraer sobre especificaciones[8]. Estas fórmulas se implementan con predicados en `Prop`.

La reflexión lógica de una función es entonces el par (P, Q), su especificación, con P un predicado sobre (la reflexión lógica de) el argumento y Q un predicado sobre (las reflexiones lógicas de) el argumento y el resultado. Aquí, una vez más, se sigue la noción que sólo se necesita la especificación de una función para razonar con ella o con programas que la involucren.

Las variables libres en PCF tienen un valor asociado en el ambiente de variables durante

[8] Los autores señalan trabajo previo con respecto a la solidez y completitud de extensiones de la lógica de Hoare, notablemente el resultado de Clarke [5] sobre lenguajes con procedimientos de orden superior, recursión, alcances estáticos, variables globales y declaraciones anidadas de procedimientos para los cuales no es posible tener una lógica completa, y la solución de Damm y Josko [8] de pasar de un lenguaje de aserciones de primer orden a uno de orden superior. Solidez y completitud de la lógica para PCF no serán discutidas en este trabajo.

la ejecución, sin embargo, este ambiente no cambia en ningún momento de la ejecución, y no se introducen nuevas variables libres en el texto de un programa como se hace en otras definiciones. Las variable libres en este lenguaje son "huecos" en un programa, y la finalidad del ambiente de variables es proporcionar valores para llenar esos huecos en lugar de una expresión que pueda sustituir a la variable, esto con el propósito de tomar las variables libres (que no son valores, y que sin un ambiente de variables bloquearían la ejecución) y llevarlas a las aserciones en el nivel lógico.

También, impondremos una restricción al lenguaje, específicamente sobre los valores que pueden tener asociados las variables en el ambiente de variables pues deseamos que estos tengan el mismo tipo que la variable en el contexto de tipos. Dado un contexto de tipos Γ y un ambiente de variables E, definimos el predicado $compatible$ como:

$$compatible\ \Gamma\ E \equiv \forall x\ T\ T' v.\ E(x) = v \to \Gamma \vDash x : T \to \Gamma \vDash v : T' \to T = T' \tag{3.4}$$

Informalmente, el predicado $compatible$ nos dice que variables y valores asociados tienen el mismo tipo, con esto queremos que sea seguro reemplazar una variable libre durante la ejecución de un programa por el valor asociado en el ambiente. Esto es relevante pues si una variable fuera reemplazada por un valor arbitrario con tipo distinto durante la evaluación, es fácil ver que el lenguaje fallaría inmediatamente en tener propiedades deseables en un lenguaje de programación como *preservación de tipos* y *progreso*, aunque estas propiedades no serán demostradas en este trabajo.

Las definiciones de aserciones, de ternas y las reglas de inferencia se encuentran en el archivo `PCF_Hoare.v`, pero antes veremos algunas definiciones para reflejar tipos y valores de PCF en el nivel lógico.

3.5.1. Reflexión lógica de tipos y valores

Definimos la función `typ_refl` (archivo `PCF_Typing.v`) cuyo propósito es reflejar los tipos del nivel computacional en el nivel lógico. Dado un tipo X, su reflexión lógica, denotada $[X]$, es:

- `nat`, el tipo de los números naturales, si $X = N$,
- `bool`, el tipo de los valores booleanos, si $X = B$,
- $([U] \to$ `Prop`$) \times ([U] \to [T] \to$ `Prop`$)$, el tipo de las especificaciones de funciones con tipo $X = U \Longrightarrow T$.

PCF_Types.v

```
Reserved Notation "[ X ]".

Fixpoint typ_refl (X : typ) : Type :=
  match X with
  | N       => nat
  | B       => bool
  | U ==> T => ([U] -> Prop) * ([U] -> [T] -> Prop)
  end

where "[ X ]" := (typ_refl X).
```

Ahora definimos la función `val_refl` (archivo `PCF_ValRefl.v`) que refleja los valores del nivel computacional en el nivel lógico, los cuales son habitantes de la reflexión lógica de su tipo. Dado un valor v, un tipo T y un contexto de tipos Γ tal que $\Gamma \vDash v : T$, la reflexión lógica de v es un habitante de $[T]$. La reflexión lógica de valores se define como:

- n si $v = n$,
- *true* si $v = tt$,
- *false* si $v = ff$,
- (P, Q) si $v = \lambda_{(U,T)(P,Q)}.t$,
- (P, Q) si $v = \mu_{(U,T)(P,Q)}.t$.

PCF_ValRefl.v

```
Definition val_refl (v : val) (T : typ) (G : ctx) : G |= v ~: T -> [T].
Proof.
  intros X;       (* X es una prueba de G |= v ~: T.       *)
  destruct v;     (* Análisis de casos sobre el valor v.   *)
  inversion X;    (* Nos da una nueva hipótesis X0, donde  *)
  inversion X0;   (* X0 es una prueba de G |= v ≈: T.       *)
  subst; simpl.
  - apply n.      (* v es un número natural n,
                     regresamos n.                         *)
  - apply true.   (* v es tt, regresamos true.             *)
  - apply false.  (* v es ff, regresamos false.            *)
  - apply (P,P0). (* v es una abstracción lambda,
```

```
                    regresamos su especificación.        *)
  - apply (P,P0). (* v es una abstracción fix,
                    regresamos su especificación.        *)
Defined.
```

3.5.2. Aserciones y Ternas de Hoare

Las aserciones describen la relación que hay entre las variables libres de un programa PCF y el valor resultado de la evaluación de este. La precondición o *rely* describe propiedades sobre las variables libres de un programa que han de ser ciertas antes de evaluarlo, para ello nos referimos al ambiente

PCF_Hoare.v

```
Definition rely := eval_env -> Prop.
```

mientras que la poscondición o *guarantee* habla sobre (la reflexión lógica de) el resultado de la evaluación

PCF_Hoare.v

```
Definition guarantee := forall T, [T] -> Prop.
```

Como el tipo lógico del valor depende del tipo computacional que este tiene, `guarantee` se define como un tipo producto dependiente.

Definimos el predicado *compatible* (ecuación 3.4),

PCF_Hoare.v

```
Definition compatible (G : ctx) (E : eval_env) :=
  forall x T T' v,
    binds x v E ->
    G |= trm_fvar x ~: T ->
    G |= v ~: T' ->
    T = T'.
```

También definimos la preservación de tipos

$$\text{Si } \Gamma \vDash t : T \text{ y } E \Vdash t \Downarrow v \text{ entonces } \Gamma \vDash v : T$$

como un axioma, esta propiedad será necesaria para definir las ternas de Hoare para PCF.

PCF_Hoare.v

```
28 Axiom preservation : forall G E t v T,
29   G |= t ~: T ->
30   E ||- t ⇓ v ->
31   G |= v ~: T.
```

Esta propiedad la declaramos como axioma para no tener que demostrarla, la razón detrás de esta decisión es que tratar de demostrar la preservación de tipos en nuestra implementación de PCF conllevaría considerable trabajo pues usualmente esto requiere demostrar propiedades auxiliares sobre la relación de tipificado y el *Lema de sustitución*[9]. Todo esto nos desviaría de la meta principal de este capítulo que es definir la semántica axiomática para PCF, no formalizar la metateoría de PCF.

La idea general detrás de una terna de Hoare

$$\{P\}\ t\ \{Q\}$$

para el lenguaje PCF es "si el término t se ejecuta cuando se satisface P, entonces el valor resultante satisface Q", donde la precondición o `rely` P habla de las variables libres en t y la poscondición o `guarantee` Q habla del valor resultante de la evaluación. Sin embargo, la definición e implementación de PCF son más complicadas que en el lenguaje funcional simple y hay detalles que es necesario esclarecer.

Primero, Q habla sobre la reflexión lógica del valor v resultado de la evaluación, que (asumiendo se preservan los tipos) tiene tipo $[T]$, con T el tipo computacional de t bajo un contexto de tipos Γ, con lo que de alguna manera la definición de terna tendrá que involucrar el juicio de tipificado $\Gamma \vDash t : T$, pues `guarantee` es un tipo dependiente, junto con el axioma de preservación de tipos para tener el juicio $\Gamma \vDash v : T$. Segundo, la relación de evaluación depende de un ambiente de variables E, que debe ser compatible con el contexto Γ.

Recapitulando, para la definición de terna requerimos los siguientes puntos:

1. El contexto de tipos Γ,
2. El ambiente de variables E de la evaluación tal que $compatible\ \Gamma\ E$,
3. El tipo T de t y el juicio de tipificado $\Gamma \vDash t : T$,
4. El valor v tal que $E \Vdash t \Downarrow v$, y
5. La reflexión lógica de v.

[9] Véanse las conclusiones para una discusión al respecto.

Anotaremos las ternas de Hoare con el contexto Γ

$$\Gamma \vdash \{P\}\ t\ \{Q\}$$

en caso de ser necesario extender el contexto al abrir una expresión con una o más variables.

Entonces definimos la terna de Hoare formalmente como

$$\begin{aligned}\Gamma \vdash \{P\}\ t\ \{Q\} \equiv \forall E.\ \mathit{compatible}\ \Gamma\ E \to \\ \forall T\ v.\ \Gamma \vDash t : T \to E \Vdash t \Downarrow v \to \\ P\ E \to Q\ T\ \mathbf{v}\end{aligned} \tag{3.5}$$

La reflexión lógica de v, escrita **v** en 3.5, se obtiene al utilizar el axioma `preservation` para obtener una prueba de $\Gamma \vDash v : T$, y entonces aplicar la función `val_refl` para obtener el término **v** con tipo $[T]$.

PCF_Hoare.v

```
36 Definition anotacion (G : ctx) (P : rely) (t : trm) (Q : guarantee) : Prop
       :=
37   forall E, compatible G E ->
38     forall T v (p : G |= t ~: T) (q : E ||- t ⇓ v), P E ->
39       Q T (val_refl (preservation p q)).
40
41 Notation "G |- {{ P }} t {{ Q }}" := (anotacion G P t Q)
42   (at level 90, t at next level) : pcf_hoare_scope.
```

Es importante aclarar aquí que dado que `preservation` es un axioma, el término

```
(preservation p q) : G |= v : T
```

no tiene contenido computacional, y en consecuencia

```
val_refl (preservation p q) : [T]
```

tampoco. Esto probará ser fuente de dificultades más adelante.

3.5.3. El sistema de inferencia

En general, la estructura de las reglas para el sistema de inferencia para este lenguaje no cambia mucho a comparación de aquellas presentadas para el lenguaje funcional simple (figura 2.5). Aquí tenemos reglas para constantes numéricas

PCF_Hoare.v

```
46 Theorem hoare_num : forall (G : ctx) (n : nat) (Q : guarantee),
47     G |- {{ fun _ => Q N n }} trm_num n {{ Q }}.
```

y para las constantes booleanas

PCF_Hoare.v

```
Theorem hoare_tt : forall (G : ctx) (Q : guarantee),
    G |- {{ fun _ => Q B true }} tt {{ Q }}.
```

PCF_Hoare.v

```
Theorem hoare_ff : forall (G : ctx) (Q : guarantee),
    G |- {{ fun _ => Q B false }} ff {{ Q }}.
```

que siguen la misma estructura que las reglas HoareNum y HoareBool del lenguaje funcional simple. Lo mismo se puede decir de la regla para expresiones condicionales

PCF_Hoare.v

```
Theorem hoare_if : forall G t1 t2 t3 P Q R,
    G |- {{ P }} t1 {{ R }} ->
    G |- {{ fun E => P E /\ R B true }} t2 {{ Q }} ->
    G |- {{ fun E => P E /\ R B false }} t3 {{ Q }} ->
    G |- {{ P }} trm_if t1 t2 t3 {{ Q }}.
```

con respecto a la regla HoareIf.

En cuanto a las reglas para operadores aritméticos y el test **iszero**, en todos estos casos se utiliza lo que se puede concluir de las subexpresiones de un término para concluir la poscondición de este último.

PCF_Hoare.v

```
Theorem hoare_plus : forall G t1 t2 P (Q : guarantee) Q1 Q2,
    G |- {{ P }} t1 {{ Q1 }} ->
    G |- {{ P }} t2 {{ Q2 }} ->
    (forall n m, Q1 N n -> Q2 N m -> Q N (n + m)) ->
    G |- {{ P }} trm_plus t1 t2 {{ Q }}.
```

PCF_Hoare.v

```
Theorem hoare_prod : forall G t1 t2 P (Q : guarantee) Q1 Q2,
    G |- {{ P }} t1 {{ Q1 }} ->
    G |- {{ P }} t2 {{ Q2 }} ->
    (forall n m, Q1 N n -> Q2 N m -> Q N (n * m)) ->
    G |- {{ P }} trm_prod t1 t2 {{ Q }}.
```

PCF_Hoare.v

```
Theorem hoare_suc : forall G t1 P (Q : guarantee) Q',
    G |- {{ P }} t1 {{ Q' }} ->
    (forall n, Q' N n -> Q N (S n)) ->
    G |- {{ P }} trm_suc t1 {{ Q }}.
```

PCF_Hoare.v

```
Theorem hoare_pred : forall G t1 P (Q : guarantee) Q',
    G |- {{ P }} t1 {{ Q' }} ->
    (forall n, Q' N n -> Q N (pred n)) ->
    G |- {{ P }} trm_pred t1 {{ Q }}.
```

PCF_Hoare.v

```
Theorem hoare_iz : forall G t1 P (Q : guarantee) Q',
    G |- {{ P }} t1 {{ Q' }} ->
    (Q' N 0 -> Q B true ) ->
    (forall n, Q' N (S n) -> Q B false) ->
    G |- {{ P }} trm_iz t1 {{ Q }}.
```

Dado que `guarantee` es un tipo dependiente, las implicaciones de la forma $Q_1 \to Q_2 \to Q$ y $Q' \to Q$, que involucran las poscondiciones para las subexpresiones y la poscondición de la terna en la conclusión de las reglas, reciben como argumentos el tipo y el valor resultado de la expresión correspondiente. En el lenguaje funcional simple, las poscondiciones solo se referían al valor resultado de la evaluación (figureas 2.5c y 2.5d), y se sabía si este era un número natural o un booleano dependiendo el tipo de terna, y en consecuencia, si la poscondición era de tipo `guarantee` o `guarantee_bool`.

También tenemos la regla de consecuencia

PCF_Hoare.v

```
Theorem hoare_consequence : forall G t (P : rely) (Q : guarantee) P' Q',
    G |- {{ P' }} t {{ Q' }} ->
    (forall E, P E -> P' E) ->
    (forall T v, Q' T v -> Q T v) ->
    G |- {{ P }} t {{ Q }}.
```

Las demostraciones de las reglas que se han listado hasta ahora han sido admitidas (salvo la regla `hoare_consequence`) dado que la meta a resolver en Coq tiene la forma

```
Q T (val_refl (preservation Htyping Heval))
```

donde la hipótesis `Htyping` es el juicio de tipificado del programa anotado

```
Htyping : G |= t ~: T
```

y la hipótesis `Heval` es la relación de evaluación

```
Heval : E ||- t ⇓ v
```

(ambas hipótesis se obtienen de la definición de terna como se dio en la ecuación 3.5) y como ya se había mencionado, `preservation` es un axioma y la aplicación de `val_refl` no tiene contenido computacional. Idealmente, la aplicación de `val_refl` debería reducirse a un término de tipo `[T]`, la reflexión lógica de `v`, pero Coq no puede hacer esto.

Para apreciar lo que está ocurriendo aquí, analicemos el contexto de prueba de la regla `hoare_num`

```
1 subgoal


  G : ctx
  n : nat
  Q : guarantee
  E : eval_env
  Hc : compatible G E
  Htyping : G |= n ~: N
  Heval : E ||- n ⇓ n
  HQ : Q N n
  X : typing_val G n N
  H1 : term_val n
  ============================
  Q N (val_refl (preservation Htyping Heval))
```

El término `(val_refl (preservation Htyping Heval))` tiene tipo `[N]` y debería poder reducirse a `n`, sin embargo Coq no puede aplicar la función `val_refl` porque, aunque sabe que `(preservation Htyping Heval)` es una prueba de `G |= n ~: N` (ese es su tipo), el término sintáctico no tiene contenido computacional alguno, no es un testigo y no puede operar con él. Esquivamos este obstáculo reemplazando manualmente `(val_refl (preservation Htyping Heval))` por `n` con la táctica

```
replace (val_refl (preservation Htyping Heval)) with n.
```

lo que nos deja en el siguiente contexto de prueba

```
2 subgoals

  G : ctx
  n : nat
  Q : guarantee
  E : eval_env
  Hc : compatible G E
  Htyping : G |= n ~: N
  Heval : E ||- n ⇓ n
  HQ : Q N n
  X : typing_val G n N
  H1 : term_val n
  ============================
  Q N n

subgoal 2 is:
 n = val_refl (preservation Htyping Heval)
```

La submeta actual es exactamente la hipótesis `HQ`, sin embargo, la segunda submeta no podemos demostrarla porque es exactamente la reducción que Coq no puede realizar. Entonces la demostración de la regla `hoare_num` queda incompleta y la admitimos.

Situaciones análogas evitan que podamos demostrar las demás reglas. Tentativamente este obstáculo puede ser resuelto demostrando la propiedad de preservación de tipos para PCF, pero ya se dijo previamente que tal tarea no se llevará a cabo en este trabajo, así que de momento simplemente admitimos los teoremas.

Una posible solución que podría saltar a la mente sería recurrir a axiomas que permitan resolver estas ecuaciones que Coq no puede, así como se hizo con la regla HoareFunc, que se implementó como un axioma puesto que no se puede demostrar como teorema. La idea sería siempre que haya una aplicación de la función `val_refl` a un valor (junto con su juicio de tipificado), uno de estos axiomas nos dice que esto es igual a su reflexión lógica esperada. Con esto estaríamos reemplazando la funcionalidad que deseamos implementar con `val_refl`, la reflexión lógica de valores, y en consecuencia, evitando parte del trabajo a realizar que es encajar el nivel computacional en la lógica de Coq, por lo que no recurrimos a axiomas en esta ocasión.

Un problema distinto surge al tratar de formalizar una regla para variables libres. Aquí nos interesa proponer una regla similar a la regla HoareVar para el lenguaje funcional simple,

donde la poscondición figura en la precondición. Si Q es poscondición para una variable libre x, Q habla sobre el valor v asociado a la variable x por un ambiente de variables E, además, Q habla también de (recibe como argumento) el tipo T de x según el contexto Γ. La regla para variables debe tener entonces la forma

$$\frac{}{\Gamma \vdash \{Q\ T\ v\}\ x\ \{Q\}}$$

pero recordemos que la precondición es un predicado sobre el ambiente de variables, corregimos esta regla acorde

$$\frac{}{\Gamma \vdash \{\lambda E.\ \exists v.\ E(x) = v \wedge Q\ T\ v\}\ x\ \{Q\}}$$

Aquí requerimos que la variable libre x tenga un valor v asociado en el ambiente E, por ello la cuantificación existencial.

Pero hay algo incorrecto en esta regla: v es un valor en el nivel computacional y Q habla sobre su reflexión lógica. Para esto recurrimos a la función `val_refl`, que además necesita el juicio de tipificado de v

$$\frac{}{\Gamma \vdash \{\lambda E.\ \exists v\ p.\ E(x) = v \wedge Q\ T\ (\texttt{val\_refl}\ p)\}\ x\ \{Q\}}$$

donde p es una prueba de $\Gamma \vDash v : T$.

PCF_Hoare.v

```
151 Theorem hoare_var : forall (G : ctx) (x : var) (T : typ) (Q : guarantee),
152     G |- {{ fun E => exists (v : val), exists (p : G |= v ~: T), get x E =
            Some v /\ Q T (val_refl p) }} trm_fvar x {{ Q }}.
```

En esta regla se hace presente de nuevo el problema con la reflexión lógica de valores y el axioma `preservation`, pero en este caso tenemos dos instancias. Igual que antes, tenemos una meta por resolver de la forma

```
Q T (val_refl (preservation Htyping Heval))
```

pero además tenemos entre nuestras hipótesis `Q T (val_refl p)`, la precondición de la terna, y aquí Coq tampoco es capaz de aplicar la función `val_refl`. Esta hipótesis debería ser suficiente para resolver la meta, pero una vez más admitimos la regla.

Las abstracciones lambda en PCF están anotadas con fórmulas, por lo que una terna para $\lambda_{(U,T)(P,Q)}.t$ debe adaptarse a una terna sobre el cuerpo t, abierto con una variable libre x, que utiliza estas fórmulas como precondición y poscondición

$$\frac{\begin{array}{c}\forall x \notin L.\quad \Gamma, x : U \vdash \{R \wedge P\}\ t^x\ \{Q\} \\ (P \to Q) \to S\end{array}}{\Gamma \vdash \{R\}\ \lambda.t\ \{S\}}$$

Además necesitamos saber que la especificación de la función, codificada en la implicación $P \to Q$ en el segundo antecedente de la regla, implica la poscondición S que se quiere concluir sobre la abstracción. Esta es la regla para abstracciones que aparece en [13]. Recordemos aquí que P y Q son la precondición y poscondición que carga la abstracción; son las aserciones para el cuerpo de la abstracción, no de la abstracción.

En esta regla requerimos saber que la variable x tiene un valor v asociado en el ambiente E, que es el argumento que recibe la función al ser aplicada. Podríamos incorporar esta información a la regla como sigue

$$\frac{\begin{array}{c}\forall x \notin L.\, \exists v. \quad \Gamma, x : U \vdash \{\lambda E.\, E(x) = v \wedge R\, E \wedge Pv\}\ t^x\ \{Qv\} \\ (P \to Q) \to S\end{array}}{\Gamma \vdash \{R\}\ \lambda.t\ \{S\}}(\star)$$

pero P y Q no esperan un valor v del nivel computacional, más bien la reflexión lógica de v, y para ello necesitamos una prueba de que v tiene tipo U bajo Γ. Entonces, al igual que como ocurrió con la regla para variables, necesitamos cuantificar existencialmente una prueba p de $\Gamma \vDash v : U$.

Pero hay un segundo problema aquí, y es que la variable x es fresca, por lo que no debería estar definida en el ambiente de variables E cuantificado en la terna para t^x, que además es distinto al ambiente cuantificado en la terna de la abstracción. Tenemos ahora el problema del cambio de ambiente al que nos enfrentamos en la regla HoareLet del lenguaje funcional simple, que requería hacer cambios de ambientes explícitos. Introducir más aserciones e implicaciones a la regla la haría demasiado engorrosa y poco clara, y esto sin mencionar el tercer problema presente en la regla.

Las aserciones `guarantee`, como S, son funciones dependientes con tipo $\forall X.\, [X] \to$ `Prop`, mientras que la poscondición Q de la abstracción tiene tipo $[U] \to [T] \to$ `Prop`. Si queremos usar Q en una terna, como figura en el primer antecedente de $(\star)$, es necesario "adaptarla" a una función dependiente mediante una manipulación no trivial como la siguiente

$$\lambda X : \texttt{typ}.\, \lambda \mathbf{w} : [X].\, \texttt{if}\ X = T\ \texttt{then}\ Q\mathbf{vw}\ \texttt{else False}$$

donde "envolvemos" a Q en una función que recibe un tipo `typ` X arbitrario y un habitante $\mathbf{w}$ de $[X]$, y si el tipo X es T (el tipo de regreso de la abstracción $\lambda.t$), regresamos la proposición $Q\mathbf{vw}$, es decir, Q aplicada a la reflexión lógica de v y la reflexión lógica $\mathbf{w}$ del valor w resultante de la ejecución de t^x. En caso contrario, t^x se evalúo a un valor w que no tiene tipo T (algo que no debería ocurrir si la abstracción está correctamente tipificada) y regresamos `False`.

Debe ser claro en este momento que ya no es plausible diseñar una regla útil para anotar abstracciones. El caso de las abstracciones fix será igual, o más complicado dado que es necesario abrir el cuerpo de la abstracción con dos variables.

En cuanto a una aplicación de función $t_1\ t_2$, no tiene mucho sentido tratar de presentar una regla puesto que las reglas para abstracciones fueron un fracaso, pero idealmente la regla tendría la forma

$$\frac{\Gamma \vdash \{P\}\ t_1\ \{R\} \qquad \Gamma \vdash \{P\}\ t_2\ \{S\} \qquad R \wedge S \to Q}{\Gamma \vdash \{P\}\ t_1\ t_2\ \{Q\}}$$

que es la regla para aplicación de funciones presentada en [13].

Dado que no pudimos diseñar con éxito reglas para abstracciones lambda o fix, no tenemos un sistema de inferencia útil para PCF. La única regla que pudo demostrarse con éxito fue la regla de consecuencia, pero esto es debido a que su demostración depende más de la lógica subyacente que habitan las aserciones que de el significado formal de las ternas.

Si se desea intentar perseguir esta implementación quizá sea necesario reconsiderar la forma en que reflejamos los valores del nivel computacional en el nivel lógico, o considerar una estrategia enteramente distinta para formalizar no solo las reglas, sino las aserciones y ternas.

Se señaló previamente que los problemas que surgen debido a las aplicaciones de la función `val_refl` y el axioma `preservation` podrían resolverse demostrando el enunciado de preservación de tipos, pero de no ser el caso una ruta alternativa a seguir podría ser implementar la reflexión de valores sin depender de juicios de tipificado, aunque esta solución no es inmediatamente obvia pues la reflexión lógica de un valor v debe tener tipo $[T]$ en el nivel lógico, por lo que a primera vista es necesario tener un juicio que relaciona v con su tipo computacional T.

Otra opción sería deshacernos del juicio de tipificado embebido en la definición de terna, de hacer esto deberíamos cambiar las aserciones, específicamente la definición de *guarantee*, que depende del tipo computacional T del resultado de la evaluación, y en tal caso necesitaríamos recuperar la información de los tipos de los programas de algún lugar ya que los tipos de las reflexiones lógicas son inseparables de los tipos computacionales.

Una tercera opción, relacionada a la previa, y posiblemente la que implicaría más esfuerzo, sería implementar un lenguaje de orden superior para expresar las aserciones que nos interesan, junto con un sistema de tipos para verificar que las fórmulas de este lenguaje, reflexiones lógicas y variables estén bien tipificadas, permitiendo así extraer los juicios de tipificado (de programas) de la definición de terna, esta información ya no es necesaria pues las aserciones ya están bien tipificadas. Esta es la opción tomada en [18, 13].

Conclusiones

El lenguaje funcional simple presentado en el capítulo 2 fue intencionalmente diseñado con restricciones artificiales que reducen en gran medida su utilidad y flexibilidad, esto a comparación de lenguajes de programación funcionales modernos, pero en su sencillez encontramos la ventaja de ser un lenguaje con el cual es muy fácil trabajar. De una manera similar al artículo original de Hoare [11], donde el lenguaje imperativo estudiado es pequeño pero cuenta con estructuras de control imperativas básicas, el lenguaje funcional simple fue pensado tratando de capturar las estructuras de control funcionales y tipos de datos básicos, en aras de poder tomar las ideas que subyacen la lógica estilo Hoare presentada y puedan ser replicadas en la construcción de sistemas de inferencia para lenguajes funcionales más interesantes.

El sistema de inferencia presentado en la sección 2.4 está inspirado y se deriva de los trabajos de Yann Régis-Gianas y François Pottier en [18], y Kohei Honda y Nobuko Yoshida en [13]. En ambos trabajos se presenta una lógica estilo Hoare para lenguajes funcionales considerablemente más complejos que el lenguaje funcional simple, pero el razonamiento detrás de las reglas propuestas para las estructuras de control que comparten con el lenguaje funcional simple es esencialmente es mismo.

Se explicó con detenimiento cómo ocurre la ejecución de las estructuras de control y operadores en el lenguaje funcional simple con el propósito de "traducir" el significado proporcionado por una semántica operacional a una semántica axiomática. Aunque el resultado final fue un sistema de inferencia con un número relativamente alto de reglas (figura 2.5), al menos a comparación del caso imperativo (figura 1.2), la lógica obtenida es flexible y puede ser extendida a más tipos de programas: el lenguaje funcional simple podría extenderse con más tipos de datos y operaciones, o llevar las reglas a algún otro lenguaje con mayor funcionalidad práctica, y solo requeriríamos incrementar la lógica con nuevas reglas para describir el comportamiento de la nueva funcionalidad que sea de interés. Esta modularidad de la lógica de Hoare se debe a su naturaleza composicional y que su definición está dirigida por la sintaxis del lenguaje.

Por ejemplo, en el lenguaje funcional simple existe la posibilidad de definir y usar funciones recursivas mutuamente definidas (cosa que no fue mencionada en ningún momento para no

ensanchar la discusión más allá de lo tratable en términos de tiempo y esfuerzo). Funciones recursivas mutuamente definidas requieren añadir *una* regla al sistema de inferencia que generaliza la regla `HoareFunc` permitiendo verificar múltiples funciones simultáneamente, como lo hace Von Oheimb en [19].

En una presentación similar a la lógica de Hoare para While en el libro de Pierce [17], definimos el significado formal de un terna de Hoare apoyándonos de la semántica operacional de paso grande del lenguaje, y los axiomas y reglas de inferencia fueron expresadas como teoremas en Coq y se demostraron, salvo por la regla HoareFunc que como se mencionó tuvo que ser definida como un axioma. Las demostraciones de las reglas dan fe de la validez de las reglas en el sistema de inferencia introducido en el presente trabajo, en el sentido que su estructura preserva la verdad de los juicios (las ternas), capturando adecuadamente el razonamiento deductivo que se seguiría al razonar en papel sobre programas funcionales.

Por otra parte, el lenguaje PCF trabajado en el capítulo 3 probó ser menos noble. El sistema de inferencia para PCF sigue en gran medida el mismo diseño que la lógica del lenguaje funcional simple, pero su formalización en Coq siguiendo la metodología de [18] introdujo dificultades: el significado formal de una terna (ecuación 3.5) es una fórmula compleja y la reflexión lógica de valores implementada en la función `val_refl` en conjunción con el axioma `preservation` no nos permitió concluir las demostraciones de las reglas propuestas.

Tentativamente, las reglas admitidas en la implementación podrían ser demostradas con éxito si `preservation` fuera un teorema en vez de un axioma, lo que permitiría a Coq realizar los cómputos necesarios y resolver una meta de la forma

```
Q T (val_refl (preservation Htyping Heval))
```

con alguna de las hipótesis presentes en el contexto durante la prueba. No obstante, demostrar que el lenguaje PCF posee la propiedad de preservación de tipos requeriría considerable esfuerzo, sería necesario demostrar lemas y teoremas análogos a los que lista Charguéraud en [3] para demostrar la preservación de tipos en el cálculo lambda con tipos simples: propiedades estructurales de la relación de tipificado como el *Lema de debilitamiento*, propiedades entre la sustitución textual de variables libres y la operación abrir, y el *Lema de sustitución*. Esta tarea no es trivial y de hecho podría justificar un trabajo de tesis independiente del presente. Como se señaló en su momento, el propósito de esta tesis es diseñar axiomas y reglas de inferencia, no formalizar la metateoría de PCF, demostrar preservación de tipos queda fuera de nuestro alcance.

También, como se pudo apreciar hacia el final del capítulo 3, las reglas para abstracciones lambda, abstracciones fix y aplicaciones de funciones se volvieron en extremo complicadas y para nada claras, en marcado contraste con las reglas HoareFunc y HoareApp diseñadas para el

lenguaje funcional simple, por lo que tuvimos que dejar incompleto el sistema de inferencia para PCF, al menos hasta que podamos encontrar una manera más simple de tratar con funciones y reflexiones lógicas de valores.

Dicho todo esto, debemos considerar alternativas si perseguiremos en el futuro lograr una implementación exitosa de la lógica estilo Hoare para PCF que hemos propuesto, quizá debamos repensar la forma en que el nivel computacional se refleja en el nivel lógico, seguir una estrategia distinta a la que se eligió, redefinir las aserciones y ternas o inclusive la infraestructura básica del lenguaje (relaciones de evaluación y de tipificado), como se discutió al final del tercer capítulo. Aún hay mucho territorio que explorar, no solo en el lenguaje PCF sino en la lógica de Hoare funcional como disciplina teórica y aplicada.

Apéndice A

Representación local anónima

Es común encontrar en la literatura definiciones de lenguajes que utilizan variables para representar un elemento arbitrario de algún conjunto y utilizar ciertos nombres o símbolos para escribir estas variables. Por ejemplo, las variables proposicionales de la lógica proposicional se suelen nombrar p, q, r,... y pueden tomar el valor falso o verdadero. Esta forma de representar variables se conoce como *representación con nombres*.

Lenguajes más complejos introducen cuantificadores que ligan ocurrencias de variables en una construcción, a la cual se denomina su alcance. Por ejemplo, en una fórmula $\forall x.\ P$ de la lógica de primer orden, el cuantificador universal $\forall$ liga toda ocurrencia de la variable x en su alcance P.

Una variable libre es aquella que no está ligada por ningún cuantificador, mientras que una variable ligada es aquella que está ligada por un cuantificador; variables libres y ligadas no tienen por que ser conjuntos ajenos. Es convención que los nombres de variables ligadas pueden, en general, cambiarse en cualquier momento por un nombre conveniente sin alterar el significado de una expresión. Por ejemplo, en el cálculo lambda las expresiones $\lambda x.\lambda y.xy$ y $\lambda z.\lambda w.zw$ solo difieren en nombres de variables ligadas, pero ambos términos representan la misma función o término lambda, entonces decimos que son expresiones α-equivalentes. El hecho que distintas expresiones α-equivalentes representan un mismo término lambda al emplear la representación con nombres presenta una desventaja para una computadora pues esta no necesariamente reconoce expresiones equivalentes pero sintácticamente distintas como el mismo término.

Otra desventaja de la representación con nombres reside en la sustitución textual de variables libres, que puede no estar bien definida y requerir manipulación sintáctica de una expresión para poder realizarla. Consideremos una expresión de la forma $(\lambda x.y)[s/y]$ donde s es una expresión que contiene una variable libre con nombre x, si fuéramos a sustituir y por s dentro

de la abstracción, toda instancia de la variable libre x en s quedaría erróneamente ligada por la abstracción. Este fenómeno es conocido como *captura de variables* y surge del hecho que no hay forma de discernir una variable libre de una ligada por lo que se pueden confundir. Para evitar capturar variables es necesario renombrar la variable que liga la abstracción con un nombre fresco (i.e., un nombre no presente en $fv(s)$, el conjunto de variables libres de s), digamos z, antes de poder realizar la sustitución

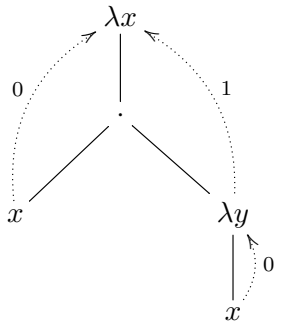
$$(\lambda x.y)[s/y] \equiv_\alpha (\lambda z.y)[s/y] = \lambda z.s$$

La elección de un nombre "fresco" suele ser en cierta medida arbitraria en pruebas en papel, y la implementación de esta manipulación sintáctica en una computadora es complicada y requiere esfuerzo extra.

Una forma de evitar captura de variables consiste en diferenciar sintácticamente variables libres de variables ligadas de tal forma que no puedan ser confundidas, evitando la necesidad de renombrar variables ligadas cuando una sustitución no esté bien definida. Si se usan nombres tanto para variables libres como para variables ligadas, esta representación se conoce como *representación local con nombres*. Sin embargo, esto no evita que los términos lambda tengan múltiples representaciones α-equivalentes.

En la *representación anónima*, o *representación con índices de De Bruijn*, las variables son representadas con índices en vez de nombres, y que apuntan a la abstracciones que las ligan. Los índices denotan el número de abstracciones (alcances) que existen entre la ocurrencia una variable en un punto dado del texto de un término y su abstracción correspondiente. Informalmente, el índice es el número de lambdas que hay que "saltar" para encontrar la lambda que liga la variable.

Esto último se puede apreciar mejor con un poco de ayuda visual, consideremos el árbol de sintaxis abstracta del siguiente término lambda $\lambda x.x(\lambda y.x)$



La representación anónima de este término es $\lambda.0\ (\lambda.1)$. La variable x en el lado izquierdo de la aplicación tiene índice 0 porque no hay que saltar ningún alcance antes de encontrar la abstracción que la liga, está directamente debajo de su abstracción. La variable x dentro de la

abstracción en el lado derecho de la aplicación tiene índice 1 porque debe saltar el alcance de la abstracción que liga la variable y antes de encontrar su abstracción.

Ventajas de la representación anónima incluyen: los términos lambda tienen una representación única (con ciertas consideraciones), no es necesario renombrar variables al realizar una sustitución textual, es fácil de tratar para una computadora. Desventajas incluyen: las variables libres al no estar ligadas por ninguna abstracción requieren un manejo no trivial para asegurar que la representación de un término sea única, las sustituciones requieren recorrer los índices en un término por lo que no es fácil de escribir y/o leer para un lector humano que trabaja en papel.

La *representación local anónima* combina las ventajas de las representaciones discutidas hasta ahora: se hace una distinción sintáctica entre variables libres y variables ligadas por lo que no se pueden confundir, al emplear índices para representar variables ligadas se evita tener múltiples representaciones α-equivalentes para un mismo término lambda, y al representar variables libres con nombres no es necesario realizar manipulaciones no triviales al texto de un término si se tuvieran que recorrer índices que representan variables libres.

En el cuadro A.1 se pueden encontrar ejemplos de términos lambda empleando las distintas representaciones discutidas aquí.

Representaciones de variables para el cálculo lambda		
Representación con nombres	$\lambda x.(\lambda y.x\ y\ z)\ w$	Nombres para variables libres y ligadas. Una variable ligada está dentro del alcance de una abstracción.
Representación local con nombres	$\lambda x.(\lambda y.x\ y\ \mathbf{z})\ \mathbf{w}$	Variables libres y ligadas se representan con distintos tipos de nombres, son sintácticamente distintas.
Representación anónima	$\lambda.(\lambda.1\ 0\ 3)\ 1$	Variables libres y ligadas se representan con índices. Una variable ligada apunta a una abstracción presente en el término.
Representación local anónima	$\lambda.(\lambda.1\ 0\ z)\ w$	Índices para variables ligadas, nombres para variables libres. Una variable ligada *debe* apuntar a una abstracción presente en el término.

Cuadro A.1: Comparación de distintas representaciones de variables

En la representación con nombres se puede identificar una variable ligada al inspeccionar el texto de un término gracias al hecho que se encuentra dentro del alcance de una abstracción lambda que liga su nombre. En la representación local con nombres y representación local anónima la distinción está presente en la gramática del lenguaje. Por ejemplo, en la gramática del cálculo lambda con la representación local anónima (figura A.1) podemos distinguir una variable ligada de una libre por ser un índice, sin embargo, existen construcciones sintácticas en esta representación que no tienen sentido.

		Términos lambda	
t	$::=$	i	*Variable ligada*
	$\mid$	x	*Variable libre*
	$\mid$	$\lambda.t$	*Abstracción lambda*
	$\mid$	$t_1\ t_2$	*Aplicación*

Figura A.1: Cálculo lambda (sintaxis empleando la representación local anónima)

En la representación anónima se puede identificar una variable ligada porque apunta a una abstracción presente en el texto, mientras que una variable libre no apunta a ninguna abstracción[1]. En la representación local anónima, un índice que no apunta a ninguna abstracción no es una variable libre, es un sinsentido que debe ser evitado.

Las construcciones sintácticas generadas por la gramática en la figura A.1 son denominadas *pretérminos*. Un pretérmino donde toda variable ligada apunta a una abstracción presente en su texto se denomina *término localmente cerrado* o simplemente *término*. Para determinar qué pretérminos son términos es necesario investigar su texto, prestando especial atención a las abstracciones.

En la representación con nombres, investigar una abstracción $\lambda x.t$ consiste en descartar λx del texto y analizar t. Lo mismo no es posible en la representación local anónima pues quitar λ de la abstracción $\lambda.t$ deja toda instancia de la variable ligada en t sin apuntar a abstracción alguna, o apuntando a una abstracción inexistente. Para analizar el cuerpo es necesario *abrir* la abstracción, esto es, reemplazar antes toda instancia de la variable ligada por una variable libre fresca.

Para abrir una abstracción con una variable libre definimos una función auxiliar $\{k \rightsquigarrow u\}\ t$ que reemplaza en t toda variable ligada con índice k con el pretérmino u, esta función se

[1]Las variables libres también se representan con índices por lo que es necesario idear un método determinista para asignarles un índice que pueda realizar una computadora. Este es el manejo no trivial de variables libres que señalamos previamente como una desventaja de esta representación. El tema no será discutido aquí pero se puede consultar en [16].

define recursivamente sobre la estructura de los pretérminos y recorre los índices cada vez que entramos al alcance de una abstracción dentro de t. Esta función se define para los pretérminos del cálculo lambda como sigue

$$\begin{array}{lcl} \{k \rightsquigarrow u\}\ i & = & \begin{cases} u & \text{si } i = k \\ i & \text{e.o.c.} \end{cases} \\ \{k \rightsquigarrow u\}\ x & = & x \\ \{k \rightsquigarrow u\}(\lambda.t) & = & \lambda.\{k+1 \rightsquigarrow u\}\ t \\ \{k \rightsquigarrow u\}(t_1\ t_2) & = & (\{k \rightsquigarrow u\}\ t_1)\ (\{k \rightsquigarrow u\}\ t_2) \end{array}$$

La operación abrir se define entonces

$$t^u := \{0 \rightsquigarrow u\}\ t$$

y se utiliza para abrir el cuerpo t de una abstracción con un pretérmino u (el índice la variable ligada es 0 en t, inmediatamente dentro del alcance de la abstracción). Particularmente útil es abrir el cuerpo t de una abstracción con una variable x, para seleccionar los pretérminos que son términos y en general para razonar con t.

Seleccionamos los términos del cálculo lambda con la representación local anónima con el predicado $term$, definido como

$$\frac{}{term\ x} \qquad \frac{term\ t_1 \qquad term\ t_2}{term\ t_1\ t_2} \qquad \frac{term\ t^x}{term\ \lambda.t} \quad (x \text{ variable libre fresca})$$

La condición "x variable libre fresca" significa que el nombre x es un nombre nuevo que no ha sido utilizado en cierto contexto, en este caso, que no figura entre los nombres de variables libres ya presentes en t, es decir, $x \notin fv(t)$. Esta regla tiene una cuantificación existencial oculta: basta presentar una variable x fresca tal que t^x es un término para concluir que $\lambda.t$ es un término.

En [2] se propone una alternativa denominada *cuantificación cofinita*, que requiere que t^x sea un término para una infinidad de variables x salvo una familia finita y arbitraria de nombres L. Así, reescribimos la regla para abstracciones como

$$\frac{\forall x \notin L.\quad term\ t^x}{term\ \lambda.t}$$

La ventaja de esta nueva regla radica en que su uso como una regla de eliminación en pruebas formales en el asistente de pruebas Coq facilita el estudio y razonamiento sobre lenguajes pues la cuantificación cofinita implica principios de inducción más fuertes que la cuantificación existencial. En efecto, mientras que la hipótesis de inducción con cuantificación existencial nos dice que

$$t^x \text{ es un término para } \textit{una} \text{ variable } x,$$

la hipótesis de inducción con cuantificación cofinita nos dice que

$$t^x \text{ es un término para } \textit{infinidad} \text{ de variables } x.$$

En [3] se puede encontrar una introducción detallada a la representación local anónima con cuantificación cofinita introducida en [2], así como aplicaciones.

Bibliografía

[1] K. R. Apt y E.-R. Olderog, «Fifty years of Hoare's logic,» *Formal Aspects of Computing*, vol. 31, n.º 6, págs. 751-807, 2019.

[2] B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack y S. Weirich, «Engineering formal metatheory,» *ACM SIGPLAN Notices*, vol. 43, n.º 1, págs. 3-15, 2008.

[3] A. Charguéraud, «The locally nameless representation,» *Journal of Automated Reasoning*, vol. 49, n.º 3, págs. 363-408, 2011.

[4] ——, *TLC: a non-constructive library for Coq*. dirección: `http://www.chargueraud.org/softs/tlc/`.

[5] E. M. Clarke Jr, «Programming language constructs for which it is impossible to obtain good Hoare axiom systems,» *Journal of the ACM (JACM)*, vol. 26, n.º 1, págs. 129-147, 1979.

[6] S. A. Cook, «Soundness and completeness of an axiom system for program verification,» *SIAM Journal on Computing*, vol. 7, n.º 1, págs. 70-90, 1978.

[7] T. Coq development team, *The Coq proof assistant*. dirección: `https://coq.inria.fr/`.

[8] W. Damm y B. Josko, «A sound and relatively complete axiomatization of Clarke's language L_4,» en *Logics of Programs*, E. Clarke y D. Kozen, eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1984, págs. 161-175, ISBN: 978-3-540-38775-6.

[9] G. Dowek y J.-J. Lévy, *Introduction to the Theory of Programming Languages*, ép. Undergraduate Topics in Computer Science. Springer-Verlag London, 2011.

[10] R. W. Floyd, «Assigning Meanings to Programs,» *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics*, vol. 19, págs. 19-31, 1967.

[11] C. A. R. Hoare, «An axiomatic basis for computer programming,» *Communications of the ACM*, vol. 12, n.º 10, págs. 576-580, 1969.

[12] ——, «Procedures and parameters: An axiomatic approach,» en *Symposium on Semantics of Algorithmic Languages*, Springer, 1971, págs. 102-116.

[13] K. Honda y N. Yoshida, «A compositional logic for polymorphic higher-order functions,» en *Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, 2004, págs. 191-202.

[14] G. I. López García, *Lógica de Hoare para Programación Funcional (Código)*. dirección: `https://bitbucket.org/Gilberto-Lopez/hoarefuncional/`.

[15] H. R. Nielson y F. Nielson, *Semantics with Applications: An Appetizer*, ép. Undergraduate Topics in Computer Science. Springer-Verlag London, 2007.

[16] B. C. Pierce, *Types and Programming Languages*. The MIT Press, 2002.

[17] B. C. Pierce, A. A. de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, A. Tolmach y B. Yorgey, *Programming Language Foundations*, ép. Software Foundations series vol. 2. Libro de texto electrónico, sep. de 2020, Versión 5.8. `http://www.cis.upenn.edu/~bcpierce/sf`.

[18] Y. Régis-Gianas y F. Pottier, «A Hoare logic for call-by-value functional programs,» en *International Conference on Mathematics of Program Construction*, Springer, 2008, págs. 305-335.

[19] D. Von Oheimb, «Hoare logic for mutual recursion and local variables,» en *International Conference on Foundations of Software Technology and Theoretical Computer Science*, Springer, 1999, págs. 168-180.

[20] G. Winskel, *The Formal Semantics of Programming Languages: An Introduciton*. The MIT Press, 1993.