Introducción a la notación Z

Maximiliano Cristiá
Ingeniería de Software 1
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario
2022

Resumen En este documento se presentan dos especificaciones Z, en orden creciente de complejidad, def funcionamiento de las cajas de ahorros de un banco.

Índice

1. El entorno de funcionamiento del sistema bancario

Antes de comenzar a modelar siempre es conveniente contar con una descripción lo más completa posible del entorno físico e institucional donde se instalará el sistema que en definitiva se implementará. Esta descripción puede constar de gráficos, tablas y textos explicativos en su mayor parte informales.

A continuación se introducirán las características más importantes del entorno que rodea al sistema que más adelante se especificará. Este entorno es el mismo para todos los modelos que se incluyen en este documento.

El sistema del banco funcionará en un servidor que correrá un sistema operativo tipo UNIX. Los archivos donde se guardarán los datos (cuentas corrientes, plazos fijos, etc.) serán archivos de texto con alguna estructura tabular. En ningún caso se modelará el acceso a los archivos: se asume que esa información está disponible para el programa. Los empleados del banco accederán al sistema por medio de terminales seriales en modo texto. Por simplicidad se asume que no es necesario que los empleados ingresen al sistema presentando usuario y contraseña: simplemente al encender sus terminales se les desplegará un menú de opciones. Se asume que cada empleado usará las opciones que necesita. Cada opción corresponde a una tarea específica del ámbito bancario (por ejemplo, acreditar dinero en una caja de ahorro en dólares). Se modelarán cada una de estas opciones.

Como todo sistema de cómputo (o máquina de software) el sistema que aquí se modela intentará por todos los medios imitar el estado del entorno. Esto se puede explicar con el siguiente ejemplo. Un cliente del banco se acerca a una caja y solicita al cajero el depósito en su caja de ahorros de $100. El cajero le exige la correspondiente boleta de depósito (que es un formulario en papel por duplicado). El cliente entrega el dinero y la boleta. El cajero toma el dinero y la boleta; cuenta el dinero; sella el duplicado de la boleta y se lo entrega al cliente. En este momento el banco cuenta con $100 más que debe a cierto cliente. Pero esto aun no está reflejado en el sistema. El sistema aun no "sabe" de esos nuevos $100 a favor de cierto cliente. Por lo tanto, el estado del banco y el estado del sistema del banco no coinciden. Si el cajero ingresa correctamente al sistema esta nueva transacción (es decir, la suma correcta a favor del cliente correcto), entonces el estado del banco y el estado de su sistema coincidirán (si ya antes lo hacían y el sistema funciona correctamente).

2. El estilo Z: máquinas de estados jerárquicas

En su forma más compleja Z permite especificar máquinas jerárquicas de estados; en su forma más simple Z se usa para especificar máquinas de estados. Asumimos dada y conocida la definición de máquinas de estados finitas; una máquina de estados es idéntica sólo que la cantidad de estados es potencialmente infinita. En Z las máquinas de estados se describen en dos grandes fases:

1. Definición del conjunto de estados. Lo primero que se hace es definir el conjunto de estados de la máquina. Para ello se escribe un *esquema de estado* el cual contiene las variables de estado de la máquina. Cada estado de la máquina queda definido por una tupla de valores particulares que se asocia a cada variable de estado. Por lo tanto, si una de las variables de estado varía sobre los enteros, entonces la máquina tiene infinitos estados.

2. Definición de las operaciones de estado. Todas las operaciones de estado juntas definen la relación de transición de estados de la máquina. Existen dos clases de operaciones de estado: aquellas que producen una transición de estado y aquellas que sólo consultan el estado actual de la máquina. Cada operación de estado que modifica el estado describe cómo la máquina transforma uno de sus estados en otro de sus estados. Las operaciones de estado se definen dando uno o varios *esquemas de operación* para cada una de ellas.

La descripción de máquinas jerárquicas de estados se explica más adelante en este apunte.

3. Cada titular puede tener solo una caja de ahorros

En este banco cada cliente se identifica por su número de documento y puede tener sólo una caja de ahorro (es un tipo de cuenta que no admite descubierto ni se pueden librar cheques). No se guardará el historial de transacciones de las cajas; sólo se preservará el saldo de cada una. Los clientes sólo pueden extraer y depositar dinero en efectivo, y solicitar el saldo de su caja. Un cliente puede solicitar el cierre de su caja de ahorro sólo si su saldo es cero.

3.1. Tipos elementales

El hecho de que los clientes se identifiquen por medio de su número de documento es un dato irrelevante al modelo. Daría lo mismo que fuera su nombre, su huella digital o una fotografía. Por lo tanto, modelamos ese conjunto con un tipo básico:

$$[DNI]$$

De esta forma no sabemos nada sobre la estructura de los elementos de *DNI*. Sin embargo, Z nos permite armar conjuntos, secuencias y otras estructuras matemáticas con ese tipo. Además podemos distinguir un elemento de otro, saber si un DNI está o no en un conjunto de DNIs, etc.

La suma de dinero que cada cliente del banco posee en su caja (es decir, el saldo) la consideramos un número natural (recordar que no se permite tener la caja de ahorros en rojo). En Z no existen los números reales. No tiene sentido que existan pues en una computadora tampoco existen. Las computadoras a lo sumo pueden representar un subconjunto finito de los números racionales. Cualquiera de estos subconjuntos es isomorfo a un subconjunto de los números naturales. Por lo tanto, si en la realidad las sumas de dinero pertenecen a un subconjunto de los números racionales, en nuestro modelo bien pueden ser naturales pues aquellos son isomorfos a estos. Porque, justamente, nosotros estamos modelando la realidad, es decir la estamos abstrayendo; lo que significa que estamos quitando los detalles que nos parecen innecesarios. No obstante es posible que nos equivoquemos y que descubramos que hemos omitido detalles que nos impiden describir ciertas cosas del banco: en ese caso no nos queda otro camino que reformular el modelo.

En Z el único tipo básico nativo es $\mathbb{Z}$, es decir los enteros. Los naturales se pueden definir así[1]:

$$\mathbb{N} == \{n : \mathbb{Z} \bullet 0 \leq n\}$$

Siempre es conveniente utilizar nombres significativos para todos los términos formales:

$$DINERO == \mathbb{N}$$

[1] En Z/EVES ya está definidos.

3.2. El estado del banco

Ahora llegó el momento de pensar en el conjunto de estados del banco. Como cada cliente sólo puede tener una caja de ahorros es más o menos natural o intuitivo pensar que el banco establece una relación funcional entre clientes y cajas de ahorro. Por otro lado, si es lo único que hace el banco (tener cajas de ahorro para sus clientes), el banco "es" sus clientes y sus cajas de ahorro. Por lo tanto, si un cliente se va, un nuevo cliente llega, un cliente deposita o un cliente extraer dinero de su caja, el banco cambia; no es el mismo que un instante antes: no posee la misma cantidad de dinero a favor de los mismos clientes o no tiene los mismos clientes. Eso es el conjunto de estados del banco. En Z ese conjunto se escribe así:

Banco

$$ca : DNI \pfun DINERO$$

¿Por qué una función? Por lo que dijimos más arriba: hay una relación funcional entre los clientes y sus cajas de ahorro. ¿Por qué una función parcial? Porque no todos los elementos de *DNI* son clientes del banco. *DNI* representa a todos los DNIs de todos los potenciales clientes del banco, y no a todos los clientes actuales. Es muy infrecuente que una función total forme parte del estado de un sistema.

¿Por qué no una secuencia? Porque una secuencia no mantiene una relación funcional entre clientes y cajas. Porque una secuencia, para este caso, no es suficientemente abstracta: establece un orden entre sus elementos que ningún empleado del banco ve o necesita. No hay ningún papel significativo en el banco que ordene a los clientes por orden de llegada al banco.

¿Por qué no un conjunto de pares $(dni, caja)$? Porque eso sería una relación de tipo $DNI \rel DINERO$, que no es una función. En este caso es demasiado abstracta: elimina detalles importantes.

Al mismo tiempo, técnicamente, *ca* se podría haber modelado con cualquiera de las estructuras matemáticas que mencionamos. Desde un punto de vista nadie podría haber dicho que uno de esos modelos es incorrecto. Pero todos ellos tienen uno o varios problemas:

- Al ser poco abstractos, eliminan ciertas implementaciones. ¿Qué programador se hubiera atrevido a implementar *ca* como una función si el modelo indica que es una secuencia?
- Al ser demasiado abstractos el ingeniero se ve forzado a especificar más cosas, lo que implica un modelo más largo y complejo (por ejemplo, predicar sobre una relación de tipo $DNI \rel DINERO$ para indicar que es una función).
- Cualquiera de los otros modelos es poco "elegante"[2].
- De hacerlo en un examen el alumno se vería privado de valiosos puntos necesarios para aprobarlo (por lo expresado en los puntos anteriores).

El estado inicial del banco sólo merece este párrafo:

BancoInicial

Banco

$$ca = \emptyset$$

[2]Concepto tan utilizado en la ciencias duras como escasamente definido.

3.3. Las operaciones

3.3.1. Nuevos clientes

Un banco sin clientes no es un negocio muy rentable así que comenzamos por modelar la posibilidad de que una persona se transforme en cliente del banco. Una persona se transforma en cliente del banco sí y sólo sí posee una caja de ahorro en el banco. Por lo tanto, al mismo tiempo en que una persona se transforma en cliente del banco se le crea una caja de ahorro. El saldo inicial de la cuenta es cero.

La persona brinda su número de DNI para que sea posible crearle su caja de ahorros. Por lo tanto, es necesario que el sistema solicite el número de DNI. Si el número ya está en uso (es decir, esa persona ya es cliente del banco), la operación finaliza con error, caso contrario el nuevo cliente es añadido al banco.

NuevoClienteOk

$\Delta Banco$
$d? : DNI$
$rep! : MENSAJES$

$d? \notin \mathrm{dom}\, ca$
$ca' = ca \cup \{d? \mapsto 0\}$
$rep! = ok$

La definición del tipo *MENSAJES* queda como ejercicio. Por otro lado recordar que en Z las funciones son conjuntos de pares, de allí la utilización de operadores de la teoría de conjuntos.

Ahora veamos el esquema de error de la operación *NuevoCliente*. Este es el caso en que el empleado del banco ingresa un número de DNI que ya está siendo utilizado. En esta situación el sistema rechazará la solicitud de creación de caja de ahorro.

ClienteExiste

$\Xi Banco$
$d? : DNI$
$rep! : MENSAJES$

$d? \in \mathrm{dom}\, ca$
$rep! = numeroClienteEnUso$

En consecuencia la operación *total* (que es la que se debe programar) queda definida de la siguiente forma:

$$NuevoCliente == NuevoClienteOk \lor ClienteExiste$$

Una operación es total si su precondición es equivalente a *true*. Para calcular la precondición de una operación primero se deben expandir todos los esquemas incluidos. La precondición de una operación es el predicado donde sólo aparecen variables de entrada y variables de estado no primadas. En lo posible siempre se deben especificar operaciones totales. El estilo Z sugiere especificar uno o más esquemas con los casos exitosos, uno o más esquemas con los casos erróneos y luego combinarlos mediantes disyunciones para formar la operación total. Los

casos exitosos siempre "hacen un delta del estado", mientras que los casos erróneos siempre dejan el estado sin cambios. Las precondiciones de los casos erróneos son las negaciones de las precondiciones de los casos exitosos. Sin embargo, técnicamente, no siempre es simple, dado los casos exitosos, saber cuáles son los casos erróneos. En otras palabras no es trivial determinar las precondiciones de los casos exitosos. Veremos un ejemplo de esto más adelante.

Es conveniente, al menos para esta primera operación, mostrar la relación entre lo que sucede en el entorno y lo que ocurre en el sistema. Una persona se aproxima a un empleado del banco y solicita una caja de ahorros en el banco. El empleado entrega a la persona un formulario por duplicado de alta de cliente. El formulario puede contener infinidad de campos y casilleros pero para el caso el único importante es el campo etiquetado con "DNI". La persona devuelve el formulario al empleado. Hasta el momento la persona no es cliente del banco, no tiene cómo probarlo. El empleado selecciona en su terminal la opción adecuada; el sistema le presentará un formulario donde ingresar el DNI del potencial nuevo cliente. El empleado ingresa el DNI consignado en el formulario. Aquí pueden darse varias alternativas: que el empleado se equivoque al tipear el número, que la persona haya mentido o se haya equivocado, etc. Lo importante es que si el sistema determina que el DNI no existe en el banco, entonces la cuenta será creada y el empleado verá en su pantalla un mensaje del tipo "Cliente aceptado"; en ese caso sellará el duplicado del formulario y se lo regresará a la persona. En ese momento el estado del entorno y del sistema coinciden. Si el sistema determina que el DNI ya existe, retornará un mensaje adecuado, no creará la caja y el empleado devolverá el formulario sin sellar o no lo devolverá. En este caso, el estado de ambas entidades también coincide.

Ejercicio 1 *Defina el tipo MENSAJES.*

Ejercicio 2 *Escriba en C un programa que implemente la operación NuevoCliente.*

Ejercicio 3 *Respecto del ejercicio 2, ¿qué diferencias tiene con el modelo? ¿Cómo implementó el tipo DNI? ¿Su implementación es isomorfa al tipo DNI? ¿Puede garantizar que su implementación verifica la especificación? ¿Cómo lo haría?*

3.3.2. Depositar dinero en una caja de ahorro

Ahora que tenemos clientes en el banco es posible que estos comiencen a operar con sus cajas de ahorro; es decir, podrán depositar y extraer dinero y consultar el saldo. Como no se admiten cajas de ahorro en rojo, modelaremos primero la operación de depósito. Para esta operación es necesario que el empleado del banco suministre el DNI del cliente y el monto a depositar. Sólo tiene sentido depositar una cantidad no nula de dinero.

DepositarOk

$\Delta Banco$
$d? : DNI$
$m? : DINERO$
$rep! : MENSAJES$

$d? \in \mathrm{dom}\, ca$
$0 < m?$
$ca' = ca \oplus \{d? \mapsto (ca\ d?) + m?\}$
$rep! = ok$

Observar cómo se accede al saldo de la cuenta antes de efectuar el depósito para luego actualizar el saldo de esa misma cuenta.

Como de costumbre debemos especificar el caso erróneo. Este se da cuando $d?$ no corresponde a un cliente del banco o cuando $m?$ es nulo. Definiremos un esquema para cada error.

ClienteInexistente

$\Xi Banco$
$d? : DNI$
$rep! : MENSAJES$

$d? \notin \mathrm{dom}\, ca$
$rep! = clienteInexistente$

MontoIncorrecto

$\Xi Banco$
$m? : DINERO$
$rep! : MENSAJES$

$m? \leq 0$
$rep! = montoNulo$

Notar que para el primer caso erróneo no solicitamos el monto a depositar. ¿Depende la existencia del cliente del monto a depositar? Piense cómo implementaría esta operación en C (¿lo haría con dos funciones o sólo con una?) y encontrará el motivo por el cual no solicitamos el monto. Lo mismo ocurre con *MontoIncorrecto*. No siempre es necesario dividir los casos erróneos en varios esquemas aunque es conveniente pues se pueden brindar mensajes de error más significativos. Más allá de esto, la operación final queda así:

$$DepositarE == ClienteInexistente \lor MontoIncorrecto$$

$$Depositar == DepositarOk \lor DepositarE$$

Ejercicio 4 *Suponga que el banco exige que para abrir una caja de ahorro el cliente debe depositar en el acto una suma de dinero no inferior a los $200. Modele este requerimiento.*

Ejercicio 5 *Respecto del Ejercicio 4 determine los cambios que se deberían dar en el entorno para que su estado coincida con el del sistema.*

3.3.3. Extracciones limitadas por el saldo

Es el turno de permitirle a los clientes recuperar su dinero. Al igual que *Depositar*, la extracción requiere la identificación del cliente y el monto que desea retirar. En este caso el monto no sólo debe ser positivo sino que también no puede superar la cantidad de dinero disponible en la caja de ahorros. Teniendo en cuenta todas estas consideraciones el caso exitoso de la operación queda de la siguiente forma:

ExtraerOk

$\Delta Banco$
$d? : DNI$
$m? : DINERO$
$rep! : MENSAJES$

$d? \in \mathrm{dom}\, ca$
$0 < m?$
$m? \leq ca\ d?$
$ca' = ca \oplus \{d? \mapsto (ca\ d?) - m?\}$
$rep! = ok$

Dos casos erróneos son semejantes a los de la operación anterior: el número de DNI ingresado no corresponde a un cliente del banco y el monto solicitado es negativo. El tercer caso, el monto a retirar es mayor que el saldo disponible, lo tenemos que definir de la forma usual (notar que ahora sí necesitamos $d?$ pues el monto es demasiado grande respecto de una cuenta en particular):

SaldoInsuficiente

$\Xi Banco$
$d? : DNI$
$m? : DINERO$
$rep! : MENSAJES$

$m? > ca\ d?$
$d? \in \mathrm{dom}\, ca$
$rep! = noPoseeSaldoSuficiente$

De esta forma la operación se completa según el estilo Z:

$ExtraerE == ClienteInexistente \lor MontoIncorrecto \lor SaldoInsuficiente$

$Extraer == ExtraerOk \lor ExtraerE$

3.3.4. Extracciones limitadas por el BCRA

Digamos que el Banco Central de la República Argentina (BCRA) establece que no es posible realizar extracciones de más de $1000 de cualquier caja de ahorro (tenga o no saldo suficiente). Como este número es arbitrario (mañana el BCRA puede bajar el límite a $500, elevarlo a $2000 o eliminarlo) y por lo tanto conviene aislarlo lo más posible. El recurso que ofrece Z para estos casos son las definiciones axiomáticas. Una definición axiomática es semejante a un parámetro de la especificación. En otras palabras, el modelo depende de ese parámetro: si el valor del parámetro cambia, el modelo cambia. Entonces introducimos una definición axiomática para el límite a las extracciones en cajas de ahorro:

$limiteExtrCA : \mathbb{N}$

$limiteExtrCA = 1000$

Como las definiciones axiomáticas son objetos globales, la definición de la especificación de *Extraer* puede ser modificada de la siguiente manera (notar que cambiamos el nombre de la operación):

ExtraerCAOk

$\Delta Banco$
$d? : DNI$
$m? : DINERO$
$rep! : MENSAJES$

$d? \in \mathrm{dom}\, ca$
$0 < m?$
$m? \leq min\ \{ca\ d?, limiteExtrCA\}$
$ca' = ca \oplus \{d? \mapsto (ca\ d?) - m?\}$
$rep! = ok$

Ejercicio 6 *Complete la especificación de ExtraerCA con todos los esquemas de error necesarios.* **Ayuda:** *no olvide los esquemas ya definidos para Extraer.*

Ejercicio 7 *Indique los cambios que deberían hacerse si el BCRA decide eliminar el límite a las extracciones.*

Ejercicio 8 *Suponga, ahora, que el BCRA reglamenta que ningún banco puede permitir extracciones de caja de ahorro de más del 50 % del saldo. Modele este requerimiento.*

3.3.5. Solicitar el saldo

La operación para solicitar el saldo es relativamente simple pero diferente a las demás pues no modifica el estado del banco. Simplemente se solicita la identificación del cliente y se emite el saldo de la cuenta asociada, siempre y cuando el cliente sea uno válido.

PedirSaldoOk

$\Xi Banco$
$d? : DNI$
$saldo! : DINERO$
$rep! : MENSAJES$

$d? \in \mathrm{dom}\, ca$
$saldo! = ca\ d?$
$rep! = ok$

Observar que se utiliza una variable de salida para comunicar el éxito o fracaso de la operación y otra para mostrar el saldo en caso de éxito. En caso de error sólo se emitirá el mensaje apropiado:

$PedirSaldo == PedirSaldoOk \lor ClienteInexistente$

Ejercicio 9 *Modele una operación que muestre un listado con los saldos de un grupo de cajas de ahorro.*

3.3.6. Un cliente se retira del banco

Finalmente modelaremos la operación que permitirá dar de baja un cliente. Un cliente puede ser dado de baja sólo si su saldo actual es cero.

CerrarCajaOk

$\Delta Banco$
$d? : DNI$
$rep! : MENSAJES$

$d? \in \mathrm{dom}\, ca$
$ca\ d? = 0$
$ca' = \{d?\} \lhd ca$
$rep! = ok$

A esta altura los esquemas de error y la definición de la operación total deberían ser simples de deducir y modelar, por lo tanto los mostramos sin más preámbulos.

HaySaldo

$\Xi Banco$
$d? : DNI$
$rep! : MENSAJES$

$d? \in \mathrm{dom}\, ca$
$ca\ d? \neq 0$
$rep! = saldoNoNulo$

$CerrarCajaE == ClienteInexistente \lor HaySaldo$

$CerrarCaja == CerrarCajaOk \lor CerrarCajaE$

Ejercicio 10 *Modele una operación que da de baja un cierto conjunto de clientes.*

4. Cada titular puede tener más de una caja de ahorros y viceversa

En esta sección modelaremos un banco un poco más real. En este banco una caja de ahorro puede estar asociada a varios titulares (clientes), y cada titular puede tener varias cajas de ahorro. Cualquier titular puede extraer, depositar y consultar el saldo de sus cajas. Existe la posibilidad de asociar una persona con una caja de ahorros ya abierta y de eliminar un titular de una caja de ahorros existente sin cerrarla (si aun tiene algún titular). Para que una persona se convierta en titular de una cuenta existente es necesario que lo autorice uno de sus titulares. Cualquier persona puede hacer un depósito en cualquier cuenta pero solo los titulares pueden hacer extracciones de las suyas. Además, modelaremos la operación de transferencia de dinero entre cajas de ahorro. Otro requerimiento importante es que el banco desea llevar un registro de los depósitos y extracciones de cada cuenta (esto se llama historia de la cuenta). Finalmente, el banco desea registrar no sólo el DNI de cada cliente sino también su nombre completo y domicilio.

Como los requerimientos de este banco son un superconjunto de los del primer banco, cada vez que sea razonable reutilizaremos partes del modelo anterior.

4.1. Tipos elementales y el estado

Nuevamente comenzamos seleccionando los tipos elementales para nuestro modelo. Necesitamos registrar el nombre completo de cada cliente y su domicilio. Sólo registrarlos; aparentemente no hay que realizar ninguna operación compleja con estos datos. Por lo tanto, no nos interesa la estructura de los nombres ni de los domicilios. Entonces usamos dos tipos básicos:

$[NOMBRE, DOMICILIO]$

Como cada caja de ahorros puede estar asociada a más de un cliente, ya no será posible identificar una caja por el número de DNI del cliente. De esta forma, necesitamos un identificador para las cajas de ahorro; en los bancos esto se llama *número de cuenta*. Como con los nombres y domicilios, tampoco nos interesa la estructura de estos números; entonces volvemos a usar un tipo básico:

$[NUMCTA]$

Ejercicio 11 *Enumere algunos escenarios en los cuales haber elegido tipos básicos para los nombres y los números de cuenta sea un error. Para cada escenario, defina los tipos que enmendarían el error.*

Una de las decisiones fundamentales, dados los nuevos requerimientos, es determinar cómo se define el estado del sistema, lo que implica elegir la forma de relacionar entre sí:

- Los nombres y domicilios de los clientes
- Los números de DNI de los clientes
- Los números de cuenta
- El saldo de la cuenta
- La historia de la cuenta

Con requisitos de esta complejidad no hay una única solución ni existe la mejor solución. Varias soluciones posibles tienen ventajas y desventajas. Algunas más que otras. Varias de ellas son más o menos equivalentes. Es conveniente, pues, elegir una solución entre ese grupo de las mejores. Cualquier solución debe tener en cuenta al menos los siguientes criterios:

- Se debe poder expresar todos los requerimientos
- La especificación debe ser lo más simple posible
- El modelo debe ser suficientemente abstracto como para no invalidar tempranamente implementaciones razonables

Un dato importante a tener en cuenta es que los esquemas definen tipos. Por ejemplo, el siguiente esquema:

$$
\begin{array}{|l}
\hline
A \\
x : X \\
y : Y \\
\hline
\end{array}
$$

define el tipo A por lo cual es posible escribir cosas como:

$$B == [a : \mathbb{P}\, A; f : A \pfun \mathbb{Z} \mid a = \mathrm{dom} f]$$

Para modelar el estado del banco ante estos nuevos requerimientos veremos un par de alternativas. La primera consiste en definir un esquema que reúna la información propia de una caja de ahorros, por un lado, y la de un cliente, por el otro:

CajaAhorros
$saldo : DINERO$
$his : \text{seq}\,\mathbb{Z}$

Cliente
$nom : NOMBRE$
$dir : DOMICILIO$

Para registrar la historia de la cuenta usamos una secuencia (y no una función) de enteros (y no de *DINERO*) porque: (a) es necesario preservar el orden en que se hicieron los depósitos y las extracciones y (b) los depósitos se registrarán como números positivos y las extracciones como números negativos (y *DINERO* son solo números positivos).

Entonces, una de las posibilidades de definir el estado del banco es la siguiente:

*Banco*2
$ca : NUMCTA \pfun CajaAhorros$
$clis : DNI \pfun Cliente$
$tits : DNI \rel NUMCTA$

Como cada cliente puede ser el titular de varias cajas de ahorro y cada caja de ahorro puede tener varios titulares, entonces no existe más la posibilidad de una relación funcional entre cliente y cuenta. Este modelo es suficientemente abstracto, permite modelar los requerimientos solicitados y es bastante simple e intuitivo. Sin embargo, algunos predicados pueden ser más complejos que si el estado del banco se define así:

*Banco*3
$clis : DNI \pfun NOMBRE \times DOMICILIO$
$ca : NUMCTA \pfun DINERO \times \text{seq}\,\mathbb{Z}$
$tits : DNI \rel NUMCTA$

pero al mismo tiempo esta alternativa hará que otros predicados sean más complejos que en la primera alternativa. Otra forma más de representar el estado podría ser la siguiente:

*Cliente*2
$nom : NOMBRE$
$dir : DOMICILIO$
$dni : DNI$

*Banco*4
$ca : NUMCTA \pfun CajaAhorros2$

*CajaAhorros*2
$saldo : DINERO$
$his : \text{seq}\,\mathbb{Z}$
$tits : \mathbb{P}\,Cliente2$

la cual es tan abstracta como *Banco*2 pero agrupa los datos de una forma, tal vez, no del todo natural (¿por qué los clientes dentro de las cajas de ahorro y no las cajas de ahorro dentro de los clientes?). Por otro lado, como en cualquiera de los casos anteriores, esta alternativa hará que algunas partes de la especificación sean más simples mientras que otras sean más complejas de expresar.

En este apunte de clase optaremos por describir el modelo utilizando la primera definición (*Banco*2) sólo por motivos pedagógicos (da lugar a la introducción de varios temas importantes de modelado y Z) y en los ejercicios se pedirá al alumno describir un modelo equivalente usando las otras dos definiciones (*Banco*3 y *Banco*4).

4.2. Operaciones utilizando promoción de operaciones

Sólo daremos la especificación de algunas operaciones, las restantes quedarán como ejercicios para el alumno. Para especificar estas operaciones utilizaremos una técnica muy poderosa y algo compleja denominada *promoción de operaciones*. Por un lado tenemos el esquema *CajaAhorros* y por el otro este esquema es parte de *Banco*2. Cuando esto ocurre decimos que tenemos un *modelo jerárquico* o una *especificación de máquinas de estados jerárquicas*. El modelo es jerárquico porque un esquema de estados (*CajaAhorros*) forma parte del estado de otro sistema (*Banco*2). Cuando se da esta situación es muy conveniente definir primero operaciones para la máquina de estados más pequeña y luego utilizar estas para definir las operaciones del modelo de mayor nivel.

Comenzamos, entonces, por definir operaciones para *CajaAhorros*. Definimos la operación de depósito (como en el modelo anterior: primero los casos exitosos y luego los erróneos):

CADepositarOk

$\Delta CajaAhorros$
$m? : DINERO$
$rep! : MENSAJES$

$m? > 0$
$saldo' = saldo + m?$
$his' = his \frown \langle m? \rangle$
$rep! = ok$

CAMontoIncorrecto

$\Xi CajaAhorros$
$m? : DINERO$
$rep! : MENSAJES$

$m? \leq 0$
$rep! = montoNulo$

$$CADepositar == CADepositarOk \lor CAMontoIncorrecto$$

Antes de continuar será muy ilustrativo analizar varios aspectos de la operación que hemos definido.

- Notar que no hemos solicitado el número de caja de ahorro. No lo hacemos pues *CajaAhorros* por sí representa todos los estados posibles de una única caja de ahorros. Por lo tanto, no tiene sentido solicitar el número: siempre se trata de la misma y única caja de ahorros.
- Sin referencia a los titulares de la caja de ahorros. Hasta este momento trabajamos con una caja de ahorros sin relación con sus titulares. En consecuencia cualquier operación se hará sin referencia a sus posibles dueños. Podría decirse que es una caja de ahorros en el limbo.
- Nombre del esquema. Precedemos el nombre de los esquemas con *CA* para distinguir esta operación de depósito de la que definiremos para el sistema *Banco*2.

Ejercicio 12 *Definir el estado inicial para CajaAhorros.*

Ejercicio 13 *Definir las operaciones de extracción y consulta de saldo. Seguir el patrón para los nombres de los esquemas.*

Ahora viene la parte complicada (pero a la vez la técnica que nos ayudará a armar modelos complejos en base a reutilizar modelos más simples). Tenemos que usar estas operaciones para definir operaciones del banco y no de una caja de ahorros en el limbo. Vamos a modelar la operación de depósito.

DepositarOk

$\Delta Banco2$
$CADepositarOk$
$n? : NUMCTA$

$n? \in \operatorname{dom} ca$
$ca\ n? = \theta CajaAhorros$
$ca' = ca \oplus \{n? \mapsto \theta CajaAhorros'\}$
$clis' = clis$
$tits' = tits$

Lo primero que queremos remarcar es que esta operación se define en términos de la operación de menor nivel (ver la inclusión de esquemas). Ahora es turno de explicar el significado del nuevo operador θ. Este operador construye instancias de un tipo[3]: θA construye una instancia del tipo A. Pero, ¿qué forma tienen las instancias del tipo A? Son tuplas o registros. Si A es:

A

$x : X$
$y : Y$

entonces las instancias de A son registros de la forma:

$$\langle x : cte\_tipo\_X;\ y : cte\_tipo\_Y \rangle$$

donde cte_tipo_X y cte_tipo_Y son constantes de tipo X e Y, respectivamente. Esto se interpreta diciendo que la variable x vale cte_tipo_X y lo mimo para y. Por lo tanto, θA construye tuplas de esa forma.

Ahora la pregunta es, ¿exactamente *qué* instancia construye θ? En otras palabras, ¿cuáles son exactamente las constantes que se "asignan" a cada variable? Se asignan cualesquiera constantes que cumplan ciertas condiciones o propiedades. Es decir, una expresión θ no dice qué instancia en particular se crea, sino que da las reglas para crearla. Pero entonces, ¿cuáles son esas condiciones o propiedades? Hay que buscarlas en el esquema donde aparecen las expresiones θ y son:

- Todos los predicados del esquema en los cuales aparecen las expresiones θ.
- Para las expresiones de la forma θA, todos los predicados donde aparecen libres las variables que se declaran en A.

[3]Siempre se utiliza para los tipos esquema; es decir los tipos que definen los esquemas.

- Para las expresiones de la forma $\theta A'$, todos los predicados donde aparecen libres y **primadas** las variables que se declaran en A.
- Para encontrar esos predicados muchas veces es necesario expandir esquemas.

Veamos con detalle el caso del esquema *DepositarOk*. Comencemos por la expresión $\theta CajaAhorros$; esta expresión aparece en la proposición:

$$ca\ n? = \theta CajaAhorros$$

Por lo tanto, la operación *DepositarOk* representa un cambio de estado únicamente cuando la instancia generada por $\theta CajaAhorros$ es idéntica a $ca\ n?$; es decir, cuando se dan estas condiciones:

$$(ca\ n?).his = (\theta CajaAhorros).his \land (ca\ n?).saldo = (\theta CajaAhorros).saldo$$

En otras palabras, la operación tiene sentido o tiene sentido analizar la operación únicamente en ese caso. Puede decirse que el operador θ generará sí o sí una instancia que cumpla esa condición. Por otro lado, hay predicados en los cuales las variables de *CajaAhorros* aparecen libres (expandir esquemas):

$$saldo' = saldo + m? \land his' = his \frown \langle m? \rangle$$

Entonces, la expresión $\theta CajaAhorros$ genera una instancia de tipo *CajaAhorros* tal que

- Es idéntica a $ca\ n?$ **y**
- Su saldo más $m?$ coincide con el saldo de la instancia generada por $\theta CajaAhorros'$.
- Su historia concatenada con $\langle m? \rangle$ coincide con la historia de la instancia generada por $\theta CajaAhorros'$.

Notar cómo se relacionan las instancias generadas por $\theta CajaAhorros$ y $\theta CajaAhorros'$. Además notar que el predicado:

$$ca' = ca \oplus \{n? \mapsto \theta CajaAhorros'\}$$

no alcanza para determinar el valor que tomará cada una de las variables de *CajaAhorros*′ razón por la cual es necesario el predicado $ca\ n? = \theta CajaAhorros$[4].

Ejercicio 14 *¿Por qué no incluímos m? como variable de entrada y rep! como variable de salida? Justifique.*

Ejercicio 15 *Modele la operación total Depositar; es decir, describa los casos erróneos.*

Ejercicio 16 *Defina el estado inicial del banco usando el esquema de estados Banco3.*

Ejercicio 17 *Modele la operación de depósito pero usando el esquema de estados Banco3. ¿Necesita usar promoción de operaciones?*

Ejercicio 18 *Defina es el estado inicial del banco usando el esquema de estados Banco4.*

[4]Si el esquema no incluyera esa precondición, no significaría que la operación esté mal definida, sólo sería no determinística. Sin embargo, se intenta describir modelos determinísticos, al menos cuando están a punto de ser entregados al equipo de programación.

Ejercicio 19 *Modele la operación de depósito pero usando el esquema de estados Banco*4*. ¿Necesita usar promoción de operaciones?*

Habiendo modelado una operación por medio de promoción de operaciones, modelaremos otra utilizando la misma técnica. El objetivo es descubrir cierto patrón de especificación que nos permitirá reducir sensiblemente la longitud y complejidad de modelos descriptos por medio de promoción de operaciones. Comenzamos entonces por la operación de extracción a nivel de una caja de ahorros:

CAExtraerOk

$\Delta CajaAhorros$
$m? : DINERO$
$rep! : MENSAJES$

$m? > 0$
$m? \leq saldo$
$saldo' = saldo - m?$
$his' = his \frown \langle -m? \rangle$
$rep! = ok$

Ejercicio 20 *Complete la operación CAExtraer.*

Entonces la operación a nivel del banco se define, sorprendentemente, de la siguiente forma:

ExtraerOk

$\Delta Banco2$
$CAExtraerOk$
$n? : NUMCTA$
$d? : DNI$

$n? \in \mathrm{dom}\, ca$
$d? \mapsto n? \in tits$
$ca\ n? = \theta CajaAhorros$
$ca' = ca \oplus \{n? \mapsto \theta CajaAhorros'\}$
$clis' = clis$
$tits' = tits$

¿Por qué? ¿Cómo puede ser? La explicación es relativamente simple y yace en el poder expresivo del operador θ: este operador captura el cambio producido en la máquina de menor nivel y lo propaga a la máquina de mayor nivel.

Dado esto, es fácil aislar la porción común entre las operaciones. Esta porción se denomina *esquema marco de promoción*. En nuestro caso este esquema es:

CajaAhorrosABanco

$\Delta Banco2$
$\Delta CajaAhorros$
$n? : NUMCTA$

$n? \in \mathrm{dom}\, ca$
$ca\ n? = \theta CajaAhorros$
$ca' = ca \oplus \{n? \mapsto \theta CajaAhorros'\}$
$clis' = clis$
$tits' = tits$

Entonces podemos especificar las dos operaciones de manera muy simple:

$$Depositar == CajaAhorrosABanco \land CADepositar$$

$$Extraer == [CajaAhorrosABanco;\ d? : DNI \mid d? \mapsto n? \in tits] \land CAExtraer$$

Y lo mejor es que si hubiera otras operaciones que involucrasen sólo a una caja de ahorro, entonces podríamos usar el mismo esquema marco de promoción. Incluso sirve, aunque un poco artificiosamente, para la operación que solicita el saldo de una caja de ahorro:

$$PedirSaldo == CajaAhorrosABanco \land CAPedirSaldo$$

Sin embargo, es conveniente mencionar un inconveniente en especificar las operaciones como se indica más arriba. En efecto, por ejemplo, el esquema *CAExtraer* incluye un caso exitoso y uno o más casos erróneos. ¿Qué ocurre con la promoción cuando se dan los casos erróneos? Claramente, en cualquiera de los casos erróneos el estado de la *CajaAhorros* no cambia por lo que $\theta CajaAhorros = \theta CajaAhorros'$. Por lo tanto, en los predicados que involucran a *ca* y *ca*′ no habrá un cambio efectivo (se sacará y agregará la misma caja de ahorros), lo que es equivalente a decir que $ca' = ca$. En otras palabras, los casos erróneos a nivel de la máquina de menor nivel se traducen en casos erróneos a nivel de la máquina de mayor nivel. Sin embargo, darse cuenta de la existencia de esta propiedad requiere analizar la especificación lo que puede estar (o debería estar) fuera del alcance de los programadores. En consecuencia una forma alternativa de usar la promoción de operaciones es:

$$CajaAhorrosNoExiste == [\Xi Banco2;\ \Xi CajaAhorros;\ n? : NUMCTA \mid n? \notin \mathrm{dom}\, ca]$$

$$\begin{aligned} Depositar == \\ &CajaAhorrosABanco \land CADepositarOk \\ &\lor CajaAhorrosNoExiste \\ &\lor \Xi Banco2 \land CAMontoIncorrecto \end{aligned}$$

$$\begin{aligned} Extraer == \\ &(CajaAhorrosABanco \land CAExtraerOk) \\ &\lor (CajaAhorrosNoExiste \lor \ldots completar \ldots) \end{aligned}$$

Ejercicio 21 *¿Por qué decimos que esta forma de definir PedirSaldo es artificiosa? ¿Qué es lo que la diferencia de las otras dos?*

Ejercicio 22 *Modelar la operación CAPedirSaldo.*

Ejercicio 23 *Modelar el alta y la baja de una caja de ahorros. ¿Puede utilizar el esquema marco de promoción? ¿Por qué?*

Ejercicio 24 *Modele las operaciones de extracción, alta y baja de un cliente y pedido de saldo utilizando el esquema de estados Banco*3*.*

Ejercicio 25 *Modele las operaciones de extracción, alta y baja de un cliente y pedido de saldo utilizando el esquema de estados Banco*4*.*

Ejercicio 26 *En el esquema marco de promoción CajaAhorrosABanco hay una precondición para la cual no se ha escrito ningún esquema de error para ninguna de las operaciones. ¿Por qué?*

Ejercicio 27 *Falta modelar un esquema de error para la operación Extraer. Hágalo.*

4.2.1. Promoción de operaciones aplicada a varios elementos – Conjuntos por comprensión

Hasta el momento hemos aplicado la promoción de operaciones a una caja de ahorros. Por otra parte el banco puede solicitar operaciones que se realicen sobre varias cajas de ahorro. Por ejemplo, podría solicitarnos una operación que recibe un conjunto de pares ordenados $(numcta, monto)$ y que ejecute el depósito de ese *monto* en la cuenta *numcta* para cada elemento del conjunto. En este caso tenemos que promover la operación *CADepositoOK* para cada *numcta* en el conjunto de entrada. La forma de hacerlo es la siguiente:

DepositarBatchOk

$$
\begin{array}{l}
\Delta Banco2 \\
batch? : NUMCTA \pfun DINERO \\
rep! : MENSAJES \\
\hline
\dom batch? \subseteq \dom ca \\
\ran batch? \subseteq \mathbb{N}_1 \\
ca' = ca \oplus \\
\quad \{n : \dom batch?; \; CADepositarOk \mid \\
\quad\quad ca\ n = \theta CajaAhorros \land m? = batch?\ n \bullet n \mapsto \theta CajaAhorros'\} \\
tits' = tits \\
clis' = clis \\
rep! = ok
\end{array}
$$

Como podemos ver el esquema hace uso de los conjuntos por comprensión de Z. En Z estos conjuntos tienen una estructura un poco más compleja que los que se usan habitualmente en matemática. Un conjunto por comprensión de Z es de la forma $\{D \mid F \bullet P\}$, donde D es una declaración como en un esquema, F se llama *filtro* y es un predicado, y P se llama *patrón* y es una expresión. El tipo del conjunto está dado por el tipo de P. Los elementos del conjunto son todas las instancias de P tales que las variables que participan están en D y verifican F. Ver más sobre los conjuntos por comprensión en los libros de Z.

En el conjunto del esquema *DepositarBatchOk* la declaración es $n : \dom batch?; \; CADepositarOk$ lo que significa que, además de n, se declaran todas las variables declaradas en *CADepositarOk* y que el predicado de *CADepositarOk* se conjuga con el filtro $ca\ n = \theta CajaAhorros \land m? = batch?\ n$.

Es decir que podríamos haber escrito el conjunto así:

$$\{n : \mathrm{dom}\, batch?;\ m?, saldo, saldo' : DINERO;\ his, his' : \mathrm{seq}\,\mathbb{Z};\ rep! : MENSAJES \mid$$
$$m? > 0 \land saldo' = saldo + m? \land his' = his \frown \langle m? \rangle \land rep! = ok$$
$$\land\ ca\ n = \theta CajaAhorros \land m? = batch?\ n \bullet$$
$$n \mapsto \theta CajaAhorros'\}$$

De esta forma, los elementos de este conjunto son pares ordenados (n, c') donde n es un elemento de $\mathrm{dom}\, batch?$ y c' es una instancia de tipo *CajaAhorros*. La forma en que se generan estos pares ordenados es la siguiente:

1. Se instancian las variables de la declaración
2. Esas instancias deben verificar el filtro
3. Se instancia el patrón con las variables anteriores

Este proceso se repite hasta efectuar todas las instanciaciones posibles del paso 1. O sea que se procede como cuando una cuantificación universal se reemplaza por una conjunción:

$$(\forall x : F(x)) \Leftrightarrow F(x_1) \land F(x_2) \land F(x_3) \land \ldots$$

donde $x_1, x_2, x_3, \ldots$ son todos los valores posibles que puede tomar x. En el caso del conjunto por comprensión los valores posibles para las variables se obtienen de los tipos respectivos.

En consecuencia, dentro del conjunto por comprensión se aplica la operación *CADepositarOk* para cada elemento de $\mathrm{dom}\, batch$ y el resultado de cada una de esas aplicaciones es uno de los elementos del conjunto. Por ejemplo si $batch? = \{n_1 \mapsto 20, n_2 \mapsto 50\}$ el conjunto por comprensión tendrá dos elementos (asumiendo que n_1 y n_2 están en $\mathrm{dom}\, ca$): $\{n_1 \mapsto \theta CajaAhorros'_1, n_2 \mapsto \theta CajaAhorros'_2\}$, donde $\theta CajaAhorros'_1$ y $\theta CajaAhorros'_2$ se generan resolviendo el siguiente predicado.

$$m_1? > 0 \land saldo'_1 = saldo_1 + m_1? \land his'_1 = his_1 \frown \langle m_1? \rangle \land rep_1! = ok$$
$$\land\ ca\ n_1 = \theta CajaAhorros_1 \land m_1? = batch?\ n_1 \land$$

$$m_2? > 0 \land saldo'_2 = saldo_1 + m_2? \land his'_2 = his_2 \frown \langle m_2? \rangle \land rep_2! = ok$$
$$\land\ ca\ n_2 = \theta CajaAhorros_2 \land m_2? = batch?\ n_2$$

donde $CajaAhorros_1$ es el esquema *CajaAhorros* decorado con el subíndice 1, y lo mismo para $CajaAhorros_2$.

4.2.2. Notas sobre la implementación de la promoción de operaciones

Si a un programador se le entrega una especificación Z que utiliza promoción de operaciones cabe preguntarse si existe alguna forma más o menos directa de implementar tal especificación o si el programador debe entender profundamente la semántica de la promoción para luego implementarla. En nuestra opinión existe una forma bastante directa de implementación sobre todo si se piensa en un diseño orientado a objetos (tema que veremos más adelante). Supongamos que un programador recibe la especificación del banco desarrollada hasta el momento y que se le ha indicado implementarla siguiendo las ideas del diseño orientado a objetos.

En ese caso el programador define la clase `CajaAhorros` con variables miembro `his` y `saldo`, y métodos `CADepositar`, `CAExtraer`, etc. Luego define la clase `Cliente` de foma semejante.

Más tarde define la clase `Banco2` dentro de la cual se declara, entre otros, el arreglo `ca` de tipo `CajaAhorros` e indexado por los números de cuenta. Esta clase tendrá métodos `Depositar`, `Extraer`, etc. El problema se reduce entonces a ver cómo ayuda la especificación usando promoción de operaciones para implementar esos métodos[5].

En primer lugar el esquema marco de promoción *CajaAhorrosABanco* se traduce en una función privada, que podríamos llamar `marco`, de la clase `Banco2` con la siguiente signatura:

```
int marco(dni d, numcta n, dinero m, oper op)
```

donde `oper` es un tipo enumerado entre todos los nombres de los métodos de `Banco2` que comparten el mismo esquema marco de promoción. Notar que siendo `marco` una función de la clase, puede acceder de forma directa la variable `ca`. En esta función los términos θ*CajaAhorros* se traducen en variables de tipo `CajaAhorros`. En tanto que los términos θ*CajaAhorros*′ se traducen en sentencias de la forma (donde `c` es una variable de tipo `CajaAhorros`):

```
case op is
  depositar: c.CADepositar(m);
  extraer  : if d is client then c.CAExtraer(m);
  ......................
endcase
```

Entonces la implementación de, por ejemplo, `Extraer` se reduce a:

```
int Extraer(dni d, numcta n, dinero m) {return marco(d, n, m, extraer);}
```

Ejercicio 28 *Muestre detalladamente la implementación de lo antedicho en lenguaje C++.*

Ejercicio 29 *Muestre con algún detalle la forma de la implementación en caso de que exista más de un esquema marco de promoción.*

4.3. Operaciones utilizando composición de operaciones

Uno de los nuevos requerimientos sobre *Banco*2 es modelar la operación de transferencia de dinero entre dos cajas de ahorro del banco. La transferencia la inicia uno de los titulares de una caja de ahorro, se debita la suma de esa caja y luego se acredita en la caja de destino (que puede pertenecer al mismo cliente o no). En otras palabras una transferencia involucra una extracción y un depósito; claramente la transferencia implica completar ambas operaciones o no hacer ninguna (el caso en que se puede hacer el depósito pero no se puede hacer la extracción es un caso erróneo de transferencia, es decir una transferencia fallida). Ambas operaciones ya fueron definidas y por lo tanto convendría contar con alguna técnica que permita reutilizarlas. Z provee el operador $\fatsemi$, llamado *composición*, que permite secuenciar dos operaciones. De esta forma la operación *Transferir* puede ser definida utilizando este operador, aunque no de la forma más simple:

$$Transferir == TransferirOk \lor ExtraerE \lor DepositarE$$

$$TransferirOk == ExtraerOk[no?/n?] \fatsemi DepositarOk[nd?/n?, saldo_1/saldo, saldo_2/saldo', his_1/his, his_2/his']$$

[5]Queda como ejercicio ver la forma de la implementación en caso de que estos campos guarden referencias a instancias de esos tipos en lugar de las instancias propiamente dichas.

donde previamente deberíamos definir[6]:

$$ExtraerOk == [CajaAhorrosABanco; d? : DNI \mid d? \mapsto n? \in tits] \land CAExtraerOk$$

$$DepositarOk == CajaAhorrosABanco \land CADepositarOk$$

$$ExtraerE == CajaAhorrosNoExiste \lor \ldots completar \ldots$$

$$DepositarE == CajaAhorrosNoExiste \lor CAMontoIncorrecto$$

Antes de explicar la semántica del operador de composición diremos que la forma más directa de usarlo sería:

$$TransferirOk == ExtraerOk \fatsemi DepositarOk$$

pero para el caso de *TransferirOk* no sería correcto lo que está expresando pues no estaría capturando adecuadamente el requerimiento de transferencia entre cajas de ahorro (más adelante veremos por qué esto es así).

Por otro lado, conceptualmente y más allá de los tecnicismos que ya introduciremos, ¿qué significa el operador de composición? La idea conceptual detrás de este operador es definir operaciones como la composición matemática de otras dos. Simplificando la cosa, podemos pensar a las operaciones de estado como funciones que parten de un estado y retornan otro. Entonces, componer dos operaciones es aplicar la segunda sobre la salida de la primera. En otras palabras, la primera operación parte del estado s_1 y retorna el estado s_2, en tanto que la segunda operación parte del estado s_2 y retorna el estado s_3. Por lo tanto la operación compuesta parte del estado s_1 y retorna el estado s_3. En nuestro ejemplo, *Transferir* parte de un estado donde la caja *no*? tiene saldo x y la caja *nd*? tiene saldo y, y llega a un estado donde la caja *no*? tiene saldo $x - m$? y la caja *nd*? tiene saldo $y + m$? (pasando por un estado intermedio en el cual m? ya fue debitado de *no*? pero aun no fue acreditado en *nd*?).

Pasemos ahora a explicar la definición correcta de *TransferirOk*, y en consecuencia la semántica de $\fatsemi$. El término $ExtraerOk[no?/n?]$ se utiliza para renombrar o sustituir variables dentro del esquema *ExtraerOk*. Más precisamente todas las apariciones libres de la variable n? se reemplazan por *no*? (que sugiere número de cuenta de origen). Para renombrar es necesario primero expandir todos los esquemas, o distribuir el renombramiento sobre todos los operadores entre esquemas. Por lo tanto tenemos:

$$ExtraerOk[no?/n?] == (CajaAhorrosABanco \land CAExtraerOk)[no?/n?]$$

$$(CajaAhorrosABanco \land CAExtraerOk)[no?/n?] == \\ CajaAhorrosABanco[no?/n?] \land CAExtraerOk[no?/n?]$$

$$CAExtraerOk == CAExtraerOk[no?/n?]$$

[6] Lo que por otra parte sería útil haber definido previamente para especificar *Extraer* y *Depositar*.

CajaAhorrosABanco[*no*?/*n*?]

$\Delta Banco2$
$\Delta CajaAhorros$
$no? : NUMCTA$

$no? \in \mathrm{dom}\, ca$
$ca\ no? = \theta CajaAhorros$
$ca' = ca \oplus \{no? \mapsto \theta CajaAhorros'\}$
$clis' = clis$
$tits' = tits$

Ejercicio 30 *¿Por qué CAExtraerOk*[*no*?/*n*?] *es igual a CAExtraerOk?*

El renombramiento sobre el término *DepositarOk* es semejante al caso anterior así que no lo detallaremos. Conceptualmente lo que hemos hecho hasta el momento con el renombramiento es:

- Distinguir entre el número de caja de ahorros sobre el que actúa *ExtraerOk* y aquel sobre el que actúa *DepositarOk* pues, precisamente, una transferencia implica debitar dinero de una cuenta y depositarlo en otra (renombramiento de *n*? sobre *EstraerOk* y *DepositarOk*)
- Distinguir entre la caja de ahorros sobre la cual se aplica *CAExtraerOk* de aquella sobre la que se aplica *CADepositarOk*, pues claramente no es la misma y de haber mantenido los mismos nombres para las variables hubiera sido un error (renombramiento de *his* y *saldo* sobre *DepositarOk*)

Llegados a este punto tenemos todos los elementos para explicar la semántica del operador de composición. La composición de esquemas tiene sentido sólo si se trata de operaciones definidas sobre el mismo estado, es decir operaciones que relacionan las mismas variables de estado. Entonces si A y B son dos operaciones definidas sobre el esquema de estado E el cual declara las variables e_1 y e_2, la composición entre A y B se define de la siguiente forma:

$$A \fatsemi B == (A[e_1''/e_1', e_2''/e_2'] \land B[e_1''/e_1, e_2''/e_2]) \setminus (e_1'', e_2'')$$

Notar cómo el estado de llegada de A y el de partida de B se igualan por medio del renombramiento de las variables de estado. La expresión $\setminus (e_1'', e_2'')$ oculta las variables introducidas por el renombramiento, de manera tal que desaparece el estado intermedio.

Explicaremos brevemente la ocultación de variables. Si tenemos:

$$A == [x : X;\ y : Y \mid P(x, y)]$$

entonces $A \setminus (x)$ es el esquema:

$$B == [y : Y \mid \exists x : X \bullet P(x, y)]$$

es decir, se elimina la variable ocultada de la declaración de variables del esquema y se cuantifica existencialmente en el predicado.

A continuación expandimos el esquema *TransferirOk* en dos pasos para mostrar un ejemplo del predicado resultante en una composición de operaciones—esta expansión se hace solo con

fines pedagógicos. El primer esquema es el que surge de aplicar directamente la definición de $\frac{9}{9}$ y el segundo es el resultado de aplicar la regla llamada *one-point*.

TransferirOk

$$
\begin{array}{l}
ca, ca' : NUMCTA \pfun CajaAhorros \\
clis, clis' : DNI \pfun Cliente \\
tits, tits' : DNI \rel NUMCTA \\
his, his', his_1, his_2 : \seq \mathbb{Z} \\
m?, saldo, saldo', saldo_1, saldo_2 : DINERO \\
nd?, no? : NUMCTA;\ d? : DNI;\ rep! : MENSAJES \\
\hline
(\exists ca'' : NUMCTA \pfun CajaAhorros;\ clis'' : DNI \pfun Cliente;\ tits'' : DNI \rel NUMCTA \bullet \\
\quad no? \in \dom ca \\
\quad \land\ ca\ no? = \theta CajaAhorros \\
\quad \land\ d? \mapsto no? \in tits \\
\quad \land\ ca'' = ca \oplus \{no? \mapsto \theta CajaAhorros'\} \\
\quad \land\ clis'' = clis \\
\quad \land\ tits'' = tits \\
\quad \land\ nd? \in \dom ca'' \\
\quad \land\ ca''\ nd? = \theta CajaAhorros_1 \\
\quad \land\ ca' = ca'' \oplus \{nd? \mapsto \theta CajaAhorros_2\} \\
\quad \land\ clis' = clis'' \\
\quad \land\ tits' = tits'' \\
\quad \land\ m? > 0 \\
\quad \land\ m? \leq saldo \\
\quad \land\ saldo' = saldo - m? \\
\quad \land\ his_2 = his_1 \frown \langle m? \rangle \\
\quad \land\ saldo_2 = saldo_1 + m? \\
\quad \land\ his' = his \frown \langle -m? \rangle \\
\quad \land\ rep! = ok)
\end{array}
$$

TransferirOk

$ca, ca' : NUMCTA \pfun CajaAhorros$
$clis, clis' : DNI \pfun Cliente$
$tits, tits' : DNI \rel NUMCTA$
$his, his', his_1, his_2 : NUMCTA$
$m?, saldo, saldo', saldo_1, saldo_2 : DINERO$
$nd?, no? : NUMCTA;\ d? : DNI;\ rep! : MENSAJES$

$no? \in \dom ca$
$nd? \in \dom(ca \oplus \{no? \mapsto \theta CajaAhorros'\})$
$tits\ d? = no?$
$ca\ no? = \theta CajaAhorros$
$(ca \oplus \{no? \mapsto \theta CajaAhorros'\})\ nd? = \theta CajaAhorros_1$
$ca' = ca \oplus \{no? \mapsto \theta CajaAhorros'\} \oplus \{nd? \mapsto \theta CajaAhorros_2\}$
$clis' = clis$
$tits' = tits$

$m? > 0$
$m? \leq saldo$
$saldo' = saldo - m?$
$his_2 = his_1 \frown \langle m? \rangle$
$saldo_2 = saldo_1 + m?$
$his' = his \frown \langle -m? \rangle$
$rep! = ok$

Analicemos brevemente el esquema anterior para comprender su significado. El sexto término de la conjunción es el más interesante: muestra cómo funciona la composición de funciones al modificar la variable *ca* "primero" registrando la extracción de *no*? y "luego" el depósito sobre *nd*?. Los restantes términos establecen las propiedades que tendrán las instancias generadas por *CajaAhorros*, *CajaAhorros*', $CajaAhorros_1$ y $CajaAhorros_2$.

Es importante tener muy en cuenta que las expansiones de esquemas que realizamos en este apunte tienen únicamente fines pedagógicos; en otras palabras, se supone que un especificador Z con cierta experiencia no necesita expandir esquemas para conocer el significado de expresiones Z. Más concretamente, las primeras fórmulas que escribimos al comienzo de esta sección es lo único que un especificador Z debería escribir sobre la operación de transferencia. Todo lo demás son explicaciones para el alumno.

Ejercicio 31 *¿Qué ocurre si no? y nd? son iguales? ¿Es necesario que sean diferentes? Si lo es, modifique el modelo para que lo sean.*

Ejercicio 32 *Definir la operación de transferencia usando los esquemas de estados Banco*3 *y Banco*4*.*

Ejercicio 33 *Determine, justificando formalmente su respuesta, si las dos expresiones que siguen son o no equivalentes.*

$$ExtraerOk[no?/n?] \fatsemi DepositarOk[nd?/n?, saldo_1/saldo, saldo_2/saldo, his_1/his, his_2/his'']$$

$$DepositarOk[no?/n?] \fatsemi ExtraerOk[nd?/n?, saldo_1/saldo, saldo_2/saldo, his_1/his, his_2/his'']$$

Ejercicio 34 *Modele, para los tres esquemas de estado que venimos considerando, operaciones para:*

1. *Obtener todas las cuentas en las que un cliente dado es titular*
2. *Todos los titulares de una cuenta dada*
3. *Última extracción por cierre; es decir, se extrae el saldo completo de una cuenta y luego se la cierra.*
4. *Obligatoriedad de un depósito inicial; es decir, para que una persona pueda abrir una caja de ahorros debe depositar una suma de dinero en el momento de la apertura.*
5. *Eliminar un titular de una caja de ahorros, si es que hay más de uno; solo el mismo titular puede hacer esto.*
6. *Agregar un titular a una caja de ahorros; solo uno de los titulares puede hacerlo*

4.3.1. Comentarios sobre la implementación de la composición

Si al programador se le entrega, como especificación de la operación de transferencia, la expresión:

$$Transferir == TransferirOk \lor ExtraerE \lor DepositarE$$

$$TransferirOk == ExtraerOk[no?/n?] \fatsemi DepositarOk[nd?/n?, saldo_1/saldo, saldo_2/saldo, his_1/his, his_2/his''']$$

$$ExtraerOk == [CajaAhorrosABanco;\ d? : DNI \mid d? \mapsto n? \in tits]$$

$$DepositarOk == CajaAhorrosABanco \land CADepositarOk$$

$$ExtraerE == CajaAhorrosABancoE \lor (CAExtraerE1 \lor CAExtraerE2)$$

$$DepositarE == CajaAhorrosABancoE \lor CAMontoIncorrecto$$

cabe preguntarse si esto es realmente una ayuda (asumiendo que el programador está capacitado para leer especificaciones Z) o todo lo contrario. En nuestra opinión, si el programador entiende bien especificaciones Z y ya ha implementado las operaciones *Extraer* y *Depositar*, la especificación de *Transferir* da suficientes indicios de cómo programarla.

Supongamos que se está programando en C++ y que *CajaAhorros* se ha definido como una clase con métodos *CADepositar*, *CAExtraer*, etc. Más aun, digamos que *Banco*2 también es una clase con métodos *Depositar*, *Extraer*, etc. Si `b` es de tipo `Banco2`, entonces una de las posibles implementaciones de *Transferir* es:

```
int transferir(DNI d, NUMCTA no, NUMCTA nd, DINERO m)
{if (err = b.esCuenta(nd))
   if (err = b.extraer(d,no,m))
     return b.depositar(b.titular(nd),nd,m);
   else
     return err;
 else
  return  err;
}
```

donde `titular()` y `esCuenta()` son métodos de la clase `Banco2` cuyo significado es obvio.

Notar que en la implementación el complejo renombramiento deja de ser necesario en virtud de que cada instancia de tipo *CajaAhorros* está perfectamente identificada por el lenguaje de programación.

Tener en cuenta que lo antedicho es correcto siempre y cuando haya una relación apropiada entre la atomicidad de las operaciones a nivel de Z y la implementación de los métodos `extraer` y `depositar`.

Ejercicio 35 *Complete en C++ la implementación de todas las operaciones definidas sobre Banco*2*.*

5. No usar funciones booleanas

Las funciones (parciales) booleanas son funciones que parten de cierto tipo y llegan al tipo *Bool*. Por ejemplo, si modelamos una guardia de hospital donde hemos declarado el tipo (básico) *PACIENTE* podríamos definir la función booleana $atendido : PACIENTE \rightarrow Bool$ tal que si $atendido\ p$ es verdadero significa que el paciente p ya fue atendido, y si es falso significa que aun no fue atendido.

El tema es que en Z no existe el tipo *Bool* (aunque se lo podría definir como un tipo enumerado, $Bool ::= t \mid f$) y las funciones booleanas no son en absoluto necesarias, y en general son perjudiciales e indican que el especificador no domina completamente la notación Z.

Toda función booleana en Z puede (y debe) ser reemplazada por un conjunto. Por ejemplo, en el caso de la guardia del hospital podríamos definir $no\_atendido : \mathbb{P}\, PACIENTE$ de forma tal que si $p \in no\_atendido$ entonces p deberá ser eventualmente atendido lo que implicará $p \notin no\_atendido$. Cuando el paciente $p?$ llega a la guardia simplemente lo ponemos entre los pacientes no atendidos $no\_atendidos' = no\_atendidos \cup \{p?\}$; y cuando el personal médico lo atiende lo sacamos de los pacientes no atendidos $no\_atendidos' = no\_atendidos \setminus \{p?\}$.

Por lo tanto si se ven tentados a modelar algo en Z con una función booleana, piénsenlo de nuevo porque seguramente hay otra forma (mejor) de hacerlo.

No confundir el error de usar funciones booleanas con el acierto de modelar algo que tiene un comportamiento binario con un tipo enumerado de dos elementos. Por ejemplo si están modelando una puerta que puede estar abierta o cerrada es correcto hacerlo de la siguiente forma:

$$PUERTA ::= abierta \mid cerrada$$

$$Abrir == [p, p' : PUERTA \mid p = cerrada \land p' = abierta]$$

6. Propiedades del modelo e invariantes de estado

Una propiedad de un modelo es un predicado lógico que es verdadero según la definición del modelo. Por ejemplo, en el modelo del banco es verdad que cada vez que un titular deposita $2 * m?$ pesos en una de sus cajas de ahorros, es lo mismo que hacer dos depósitos de $m?$ pesos. En términos más precisos una propiedad es un teorema del modelo. En efecto, todo modelo Z define una teoría axiomática de la cual es posible obtener teoremas, es decir, verdades sobre el modelo. Es posible que el modelo sea incorrecto lo cual impedirá probar ciertas propiedades o, peor aun, se podrá probar cualquiera. Esto último ocurre cuando el modelo es inconsistente, es

decir cuando dos de sus axiomas se contradicen. Todo modelo inconsistente tiene la propiedad de que *true* = *false*, lo que implica cualquier predicado P. Por otro lado, el modelo es el punto de partida para la fase de programación y testing. En consecuencia es muy importante asegurarse que el modelo es correcto, es decir, que no presenta contradicciones.

Hacia el fin de este curso veremos parte de la batería de técnicas necesarias para asegurar lo antedicho. Por ahora, nos contentaremos con aprender a expresar uno de los tipos de propiedades más importantes y comunes: invariantes de estado. Un invariante de estado es un predicado que es verdadero en cualquier estado por el cual pasa una máquina de estados. En relación a nuestro ejemplo bancario podemos mencionar que en cualquier momento el saldo de una caja de ahorro es igual a la suma de todos los depósitos menos la suma de todas las extracciones. Sintácticamente, un invariante es un predicado en el cual las únicas variables libres son las variables de estado (no primadas).

Lo mejor o lo más correcto es pensar los invariantes antes de escribir las operaciones pues de esta forma se especificarán operaciones que hagan que esos predicados sean realmente invariantes: es decir, predicados que valen en cualquier estado. Nosotros no lo hemos hecho así porque nos parece que introducir este concepto inmediatamente después de definir el estado de un sistema puede complicar al alumno. Pero es muy importante tener en cuenta que cuanto más y más temprano se medite sobre los invariantes de un modelo, seguramente se llegará a un modelo más correcto.

Existen al menos dos formas de expresar invariantes en un modelo Z. La primera que veremos es la que se muestra en la mayoría de los libros de Z[7] mientras que la segunda es la que utilizaremos en el curso porque pensamos que tiene ciertas ventajas sobre la primera.

Tener en cuenta que en muchos casos diremos que tal o cual predicado *es* un invariante del sistema cuando en realidad deberíamos decir que *debería ser* un invariante del sistema.

De ahora en más, excepto mención en contrario, cada vez que nos refiramos al modelo del banco será sobre *Banco*2.

6.1. Invariantes por definición

La forma tradicional de Z para expresar los invariantes de un modelo es incluirlos en el esquema de estado. Veamos el siguiente ejemplo:

*Banco*2

$$ca : NUMCTA \pfun CajaAhorros$$
$$clis : DNI \pfun Cliente$$
$$tits : DNI \rel NUMCTA$$

$$\mathrm{dom}\, ca = \mathrm{ran}\, tits$$
$$\mathrm{dom}\, clis = \mathrm{dom}\, tits$$
$$\forall c : \mathrm{ran}\, ca \bullet sumSeq\ c.his = c.saldo$$

Antes de analizar el significado informal del predicado veamos cuál es su significado Z. *Banco*2 representa la totalidad de estados posibles en los que se puede encontrar el banco. Al agregar el invariante se restringen esos estados a aquellos que lo verifican; todos los otros ya no son tenidos en cuenta, el banco no podrá alcanzarlos jamás no importa lo mal especificadas

[7] No afirmo que es en todos, porque no conozco todos los libros sobre Z.

que estén las operaciones. Por este motivo es que llamamos a esta forma *invariante por definición*. En otras palabras, no es necesario probar (es decir probar un teorema) que el invariante se cumple porque se cumple por definición. Es decir, el invariante es parte de la teoría y no una consecuencia de aquella.

Ahora pasemos al predicado que hemos incluido. Los dos primeros predicados atómicos establecen la relación básica entre los dominios de las variables de estado. Claramente todo número de cuenta asociado a una caja de ahorro tiene al menos un titular, y toda persona cuyos datos están registrados en el banco es titular de al menos una caja de ahorro. El tercer predicado atómico indica que la suma algebraica de todos los movimientos registrados en la historia de cualquier caja de ahorro debe ser igual al saldo en ese momento.

Antes dijimos que un invariante del banco es que en cualquier momento el saldo de una caja de ahorro es igual a la suma de todos los depósitos menos la suma de todas las extracciones. Lamentablemente este tipo de invariantes es muy difícil de expresar en Z (a menos que se hayan registrado las operaciones como lo hicimos en este caso). El motivo fundamental es que esa clase de invariantes hace referencia a secuencias de operaciones cosa para la cual Z no está bien preparado. Además, cuando el invariante se expresa por definición, aun no están disponibles las operaciones por lo cual mal se puede hacer referencia a ellas.

Ejercicio 36 *Especifique el operador sumSeq, el cual retorna la suma de una secuencia de números enteros. ¿Qué parágrafo Z utilizará? ¿Un esquema, una definición axiomática, una definición genérica?*

6.2. Estilo de B o TLA para expresar invariantes

B es un lenguaje propuesto por J. R. Abrial luego de definir Z; y TLA es un lenguaje de especificación que veremos un poco más adelante. En estos lenguajes los invariantes se expresan de una forma que nos parece más adecuada. El caso simétrico al mostrado en la sección anterior es:

*InvBanco*2

*Banco*2

$\mathrm{dom}\, ca = \mathrm{ran}\, tits$
$\mathrm{dom}\, clis = \mathrm{dom}\, tits$
$\forall c : \mathrm{ran}\, ca \bullet sumSeq\ c.his = c.saldo$

La gran diferencia con el estilo precedente es que el predicado no está incluido en el esquema del banco y por lo tanto no restringe sus estados posibles. En consecuencia una operación mal especificada no verificará el invariante. ¿Cómo se transforma *InvBanco*2 en un invariante? Probando que lo es. Es decir, probando que cada operación lo preserva. En otras palabras, probando que cada operación que parte de un estado que verifica el invariante termina en un estado que también lo verifica. En Z:

Teorema 1 (DepositarPI)

$$InvBanco2 \land Depositar \Rightarrow InvBanco2'$$

Notar que el predicado del invariante es el mismo en cualquiera de los dos estilos.

Ejercicio 37 *Defina el mismo invariante para los esquemas de estado Banco*3 *y Banco*4*.*

Ejercicio 38 *Intente formalizar el invariante: en cualquier momento el saldo de una caja de ahorro es igual a la suma de todos los depósitos menos la suma de todas las extracciones.*

6.3. ¿Por qué elegimos el estilo usado en B o TLA?

Porque es ventajoso para el programador y obliga al especificador a escribir especificaciones más claras dado que debe hacer explícitas todas las precondiciones. Veamos el siguiente ejemplo en el cual primero especificamos el invariante según el estilo de Z y luego según el estilo de B o TLA.

Comenzamos por definir un esquema de estados que solo contiene una variable de tipo entero pero que se espera que siempre tome valores en los naturales, de allí el invariante.

H

$$x : \mathbb{Z}$$

$$0 \leq x$$

Luego definimos una operación que disminuye el valor de x en 1.

Decr

$$\Delta H$$

$$x' = x - 1$$

¿Viola *Decr* el invariante? No, es imposible porque los estados en que puede estar el sistema son únicamente aquellos que lo verifican. Esto se puede ver más claramente expandiendo *Decr*.

Decr

$$x, x' : \mathbb{Z}$$

$$0 \leq x$$
$$x' = x - 1$$
$$0 \leq x'$$

Entonces se ve que, por ejemplo, la asignación $\langle\!| x : 0, x' : -1 |\!\rangle$ no pertenece a *Decr* puesto que no verifica su predicado. En general ninguna asignación tal que el valor de la variable x' sea menor que cero puede formar parte de *Decr*. En realidad lo que ocurre es que *Decr* tiene *precondiciones implícitas*. Las precondiciones implícitas de un esquema de operación aparecen cuando el esquema de estados está restringido por un invariante. Las precondiciones implícitas son aquellas precondiciones que deben cumplir las variables para no violar el invariante, pero que no fueron explicitadas por el ingeniero. En nuestro caso *Decr* podría haberse especificado de la siguiente forma:

Decr

$$\Delta H$$

$$0 < x$$
$$x' = x - 1$$

Bien. El problema es que el programador recibe la especificación:

Decr

ΔH

$x' = x - 1$

sin la precondición implícita. Entonces, ¿qué impide que el programador entienda que el siguiente programa es una implementación correcta de la especificación de *Decr*?.

```
decr () {x = x - 1;}
```

¿Debe el programador calcular las precondiciones implícitas? No parece ser su responsabilidad porque eso no es programar, es especificar o, peor aun, es hacer análisis formal de especificaciones. ¿Debe, entonces, hacerlo el especificador? Si lo hace, ¿cómo sabe que son esas las precondiciones implícitas y no otras? La única forma es probando un teorema.

¿Por qué elegimos el estilo de B o TLA para especificar los invariantes? Porque hace explícito de una vez todo lo que queda implícito usando el otro estilo. Siguiendo a TLA hubiéramos especificado así.

H

$x : \mathbb{Z}$

InvH

H

$0 \leq x$

Decr

ΔH

$0 < x$

$x' = x - 1$

donde claramente se ve que la precondición queda explícita. Entonces, el programador, leyendo únicamente *H* y *Decr*, pues *InvH* es irrelevante para él, se ve obligado a escribir el siguiente programa.

```
decr () {if 0 < x then x = x - 1;}
```

Sin embargo, persiste aun el problema de si *Decr* verifica o no el invariante. Esto se soluciona de manera sencilla introduciendo una *obligación de prueba*, es decir un teorema que debe ser probado para asegurar que *Decr* verifica el invariante.

Teorema 2 (DecrPI)

$InvH \wedge Decr \Rightarrow InvH'$

El ejemplo desarrollado en esta sección ilustra la razón fundamental por la cual preferimos el estilo de B o TLA para codificar invariantes en un modelo Z. Al seleccionar este estilo, el ingeniero se ve obligado a escribir todas las precondiciones de las operaciones, y a hacerlo de forma tal de garantizar el invariante. Por otro lado, el programador recibe un documento que con solo leerlo le provee información suficiente como para escribir implementaciones correctas sin tener que analizar en profundidad la especificación.

Después de todo los invariantes de estado son entidades que pertenecen al mundo del modelo formal: la implementación debe verificar el modelo y el modelo debe verificar los invariantes: de esta forma la implementación verificará, también, los invariantes.

Es trabajo del ingeniero probar que las operaciones satisfacen los invariantes. Es trabajo del programador escribir un programa que verifique la especificación.

7. Normalización

Una declaración de variable como la siguiente:

$$n : \mathbb{N}$$

se dice que *no está normalizada* porque $\mathbb{N}$ no es un tipo sino un conjunto. La declaración se puede *normalizar* declarando n con tipo $\mathbb{Z}$ y conjugando $n \in \mathbb{N}$ al predicado del parágrafo Z donde n está declarada. Por ejemplo si tenemos:

UnEsquema

$\ldots$
$n : \mathbb{N}$
$\ldots$

Predicado

su versión normalizada es la siguiente:

UnEsquema

$\ldots$
$n : \mathbb{Z}$
$\ldots$

Predicado $\land\ n \in \mathbb{N}$

En este caso particular el esquema se puede normalizar *también* como sigue:

UnEsquema

$\ldots$
$n : \mathbb{Z}$
$\ldots$

Predicado $\land\ 0 \leq n$

lo cual es un poco mejor porque desaparece el conjunto infinito $\mathbb{N}$.

En consecuencia, una declaración de variable $x : A$ no está normalizada si A no es un tipo sino un conjunto. La declaración normalizada de x se obtiene sustituyendo A por su tipo contenedor y conjugando $x \in A$ al predicado del parágrafo Z donde n está declarada. A continuación damos el tipo contenedor de algunos conjuntos frecuentemente usados en declaraciones de variables.

Conjunto	Tipo
$\mathbb{N}$, $\mathbb{N}_1$, $m \mathinner{..} n$	$\mathbb{Z}$
$X \pfun Y$, $X \fun Y$	$X \rel Y$
$\mathrm{seq}\, X$	$\mathbb{Z} \rel X$

Entonces, el esquema *Banco* definido al inicio del apunte no está normalizado:

Banco

$$ca : DNI \pfun SALDO$$

Se lo puede normalizar de la siguiente forma:

Banco

$$ca : DNI \rel SALDO$$

$$ca \in DNI \pfun SALDO$$

Como podemos ver, la normalización de un esquema de estados introduce predicados (tal vez) inesperados. Como vimos en la sección anterior estos predicados son invariantes de estado por definición. En particular los podemos llamar *invariantes de tipos*. De acuerdo a la sección anterior queremos evitar los invariantes por definición tanto como sea posible debido a que introducen precondiciones implícitas. Por lo tanto, cuando normalizamos un esquema lo hacemos poniendo los invariantes de tipo en otro esquema (puede ser en el esquema de invariantes generales).

Banco

$$ca : DNI \rel SALDO$$

InvBanco

$$Banco$$

$$\forall n : \mathrm{dom}\, ca \bullet 0 \leq ca\ n$$
$$ca \in DNI \pfun SALDO$$

Es decir, pondremos los predicados generados durante la normalización en el esquema donde hemos codificado los invariantes de estado (funcionales). De esta forma deberemos descargar más obligaciones de prueba pero evitaremos las precondiciones implícitas.

8. Designaciones

Lo que sigue es la especificación del caso exitoso de una de las operaciones defindas sobre una caja de ahorros.

*MQ*1*xbOk*

$\Delta KioLac$
$x23? : DINERO$
$my0! : MENSAJES$

$x23? > 0$
$x23? \leq hg1q$
$bn' = bn$
$hg1q' = hg1q - x23?$
$my0! = e1$

¿Puede adivinar de cuál se trata? ¿Es razonable que tenga que adivinar? ¿Qué hace que la especificación anterior sea mejor que esta? En esta versión, ¿está mal la lógica? Un programador que desconoce los requerimientos específicos del banco, ¿puede saber qué está programando? Imagine que el programador se las arregla para programarla (¿es posible, podría hacerlo?), ¿con cuál opción de menú hay que asociar la ejecución del programa resultante? ¿Hay alguna forma de saberlo?

La especificación corresponde a la operación *CAExtraer*. Es el resultado de sustituir las variables y cambiar el nombre del esquema original. La lógica no está mal, sólo es difícil leerla y asociarla con el requerimiento del usuario. Toda la confusión surge de la elección de identificadores que hizo el ingeniero, pero, ¿qué obligación tiene de elegir unos nombres sobre los otros? ¿Es una restricción de Z? Es decir, ¿cambia la semántica de un esquema Z al elegir un conjunto de nombres de esquema y variables en lugar de otros? La semántica de un esquema Z está dada por el lenguaje en sí y no por su relación con los requerimientos.

Los nombres de variables y de esquemas viven en el mundo de la lógica, por eso forman un modelo (que es una abstracción de la realidad). Las necesidades de una comunidad de usuarios son reales, no *son* parte de la lógica ni *son* un modelo. Un modelo es una abstracción de la realidad, por lo tanto no *es* la realidad. Sin embargo, hacemos el modelo para luego escribir un programa que finalmente es usado en el mundo real. En consecuencia hay una relación entre el modelo y la realidad. Si esa relación se pierde, como ocurrió con la operación *MQ*1*xbOk*, se podrá escribir un programa que cumpla con lo estipulado en ese esquema pero este programa nunca podrá ser usado correctamente en la realidad: sería una casualidad que el programa correspondiente a *MQ*1*xbOk* se asocie con la opción de menú correcta pues no hay en el esquema ninguna señal que indique el requerimiento que está modelando.

Por lo tanto, resulta esencial preservar la relación entre el modelo y la realidad. Supongamos que antecedemos la definición de la operación total *MQ*1*xb* con lo siguiente:

> d es un monto de dinero $\approx$ DINERO(d)
>
> La caja de ahorros tiene número de cuenta n y saldo $s \approx KioLac(n, s)$
>
> Se extrae el monto m de la caja de ahorros comunicando el resultado r de la operación al usuario $\approx MQ1xb(m, r)$

¿Podría ligar ahora el modelo con la realidad? ¿Sabe qué requerimiento está modelando la operación $MQ1xb$? ¿Puede asociarle la opción de menú correcta?

Claramente ahora podría hacerlo. Lo que acabamos de hacer es *designar* los términos formales primitivos del modelo. Cada designación liga un término formal del modelo con un *fenómeno* de interés en el mundo real. Los fenómenos de interés son eventos, operaciones o acciones y constantes. La relación entre el término formal y el mundo real es necesariamente informal porque el mundo real lo es. El símbolo ≈ demarca la frontera entre el mundo real (a su izquierda) y el mundo formal o lógico (a la derecha). Sin esta documentación el modelo es nada más que una teoría axiomática más sin conexión con la realidad, es el trabajo de un matemático y no de un ingeniero.

Por lo tanto, toda especificación debe estar precedida por la designación de todos los términos formales. En Z estos comprenden únicamente:

- Los tipos básicos (*DINERO*). En este caso las designaciones son de la forma:
  - a es un auto ≈ AUTO(a)
  - b es un bulto ≈ BULTO(b)
  - etcétera
- Los nombres de esquemas de operaciones totales (*Extraer*)
- Los nombres de esquemas de estado (*CajaAhorros*)
- Los nombres de constantes usadas en las definiciones axiomáticas que no correspondan a operadores de alguna teoría matemática o lógica (*limiteExtrCA*)
- Los nombres de los constructores de tipos libres que no correspondan a operadores de alguna teoría matemática o lógica (*ok*)

La existencia de las designaciones no prohíbe ni desalienta el uso de nombres sugerentes para los términos designados. Una de las grandes ventajas de contar con un modelo formal de los requerimientos es que se posee un documento claro, sin ambigüedades, legible, etc. Sería una tontería empañar este logro con nombres extraños.

¿Por qué no designar los nombres de las variables de estado? Porque con designar las operaciones es suficiente. Si se designan las operaciones, los estados de partida y de llegada quedan *definidos* por aquellas.

Notar que la designación no hace ninguna referencia a los requerimientos. Se trata de dejar bien claro el vocabulario esencial del dominio de aplicación del que tratan los requerimientos. De la misma forma, la cantidad de designaciones tiene que ser la mínima posible. Por ejemplo, el estado inicial de una caja de ahorros no se designa, sino que se define (como mencionamos más arriba). Es decir, la definición del estado inicial de la caja de ahorros no debe ir precedida de algo como:

La caja de ahorros acaba de ser creada ≈ *NewKioLac*

Sino que tal definición se documenta únicamente con el esquema de estado inicial tradicional en Z.

Tener presente que las designaciones se deben documentar antes de modelar y no luego. En este apunte se procedió de esta forma para poner énfasis en Z y no comenzar con algo que es ajeno al lenguaje y propio de una metodología de desarrollo que puede aplicarse con cualquier lenguaje.

Ejercicio 39 *Escriba las designaciones de los requerimientos enunciados en la sección 3.*

Ejercicio 40 *Escriba las designaciones de los requerimientos enunciados en la sección 4.*

Ejercicio 41 *¿Necesita escribir nuevas designaciones o modificar las que escribió en el problema 40 para el modelo basando en Banco3?*