Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Verificación de correctitud para tipos de datos replicados en Coq

Tesis de Licenciatura en Ciencias de la Computación

Pablo Nicolás Gómez

Director: Christian Roldán
Codirector: Hernán Melgratti
Buenos Aires, 2021

VERIFICACIÓN DE CORRECTITUD PARA TIPOS DE DATOS REPLICADOS EN COQ

La replicación de datos es un concepto fundamental en sistemas distribuidos ya que ofrece garantías como escalabilidad y alta disponibilidad a expensas de tener una visión inconsistente del estado del sistema. Esto significa que los usuarios de dicho sistema, temporalmente, podrían percibir diferencias sobre el estado del mismo.

En particular, esta tesis se concentra en un enfoque de replicación basada en estados, donde cada réplica (o nodo) que compone al sistema, transmite su estado al resto de las réplicas con el fin de que estas puedan combinarlo con su propio estado y alcancen así un mismo estado común. La literatura propone los tipos de datos replicados o RDTs (por su acrónimo en inglés, Replicated Data Types), quienes lidean con inconsistencias temporales que puedan existir y resuelven de forma automática cuando existen conflictos entre escrituras concurrentes. Diferentes líneas de investigación han abordado el problema de especificar e implementar RDTs. Más aún, hay demostraciones manuales sobre la correcta implementación de un RDT con respecto a su especificación.

En esta tesis proponemos abordar el problema de formalizar y verificar la correcta implementación de RDTs utilizando el asistente de demostraciones Coq. Coq es un sistema formal de semidecisión de manejo de demostraciones de teoremas chequeadas por computadora. Proveemos por lo tanto, de un marco de trabajo para verificar la correctitud de RDTs de una manera computarizada, lo cual ofrece una alternativa confiable y mecánica de abordar esta tarea. Más concretamente, en esta tesis nos centramos en realizar la experiencia de formalizar en Coq una especificación y una implementación de un RDT (tomando como caso de estudio el tipo de datos: *Contador*). Para esto, (i) mostraremos cómo transformar las definiciones existentes en definiciones equivalentes en Coq. Luego, (ii) probaremos que la implementación del tipo de datos es correcta, basándonos en un resultado que establece la existencia de una relación de simulación entre la semántica operacional asociada a la especificación, y la implementación concreta del *Contador*. Finalmente, (iii) presentamos una prueba en el asistente sobre la correcta implementación del tipo de datos, mostrando para esto, la existencia de dicha relación de simulación.

Palabras claves: Replicación, Tipos de datos replicados, Correctitud de implementaciones, Simulación, Coq.

Índice general

1. INTRODUCCIÓN

1.1. Motivación

Un sistema distribuido es un conjunto de réplicas o nodos distribuidos geográficamente en lugares distintos con el fin de satisfacer requerimientos no funcionales como disponibilidad y performance. La disponibilidad (propiedad que garantiza que cada pedido será eventualmente respondido) se garantiza debido a que la falla de uno de estos nodos no afecta de manera global al sistema sino solo de manera local. Por otro lado, la cercanía física entre los clientes del servicio y el nodo más cercano provee de una performance superior en comparación a un esquema no distribuido, donde cada actualización tendría que viajar mucha más distancia para efectuarse. Otra propiedad de los sistemas distribuidos es que pueden soportar el particionado del sistema, es decir, en caso de una desconexión de una parte del sistema, las partes pueden seguir funcionando independientemente.

Desafortunadamente, el desafío se presenta cuando se intenta además que el sistema garantice consistencia fuerte. Se entiende por consistencia fuerte al hecho de que un sistema pueda lograr la ilusión de funcionar como un sistema centralizado, haciendo que todos los usuarios observen en todo momento y lugar el mismo estado del sistema. El *teorema CAP*, enunciado en [5], establece que no es posible alcanzar simultaneamente consistencia fuerte, disponibilidad y tolerancia a la partición, haciendo que se deba resignar una de las tres. En la actualidad muchos sistemas optan por priorizar la alta disponibilidad y la tolerancia a la partición para proveer una mejor experiencia al usuario, lo cual los obliga a relajar la consistencia fuerte a un estado de consistencia más débil llamado *consistencia eventual*, la cual garantiza que todas las operaciones impactarán en todos los nodos del sistema y *eventualmente* estos convergerán al mismo estado [7] (un claro ejemplo de este tipo de sistemas es la base de datos *Cassandra* [6]). Al permitir que las réplicas sean temporalmente inconsistentes se habilita a que cada una pueda resolver las demandas locales y, luego, le comunique a las otras réplicas las actualizaciones ocurridas. Es debido a esto que podrían ocurrir actualizaciones en distintas réplicas que conflictúen entre sí dado que son efectuadas sobre el mismo item del sistema; llamaremos a estas, *operaciones concurrentes*. Por ejemplo, consideremos el tipo de dato `Registro Genérico`, el cual es una celda que puede ser leída y escrita en cualquier momento. El valor leído del registro luego de dos escrituras concurrentes w_1 y w_2 puede ser indefinido (en caso de que la lectura se haga sobre una réplica que aún no ha recibido ninguna de las dos actualizaciones), puede ser el valor escrito por w_1 o puede ser el valor escrito por w_2. En situaciones de este estilo el sistema debe asegurar que luego de que todas las actualizaciones se hayan propagado a todas las réplicas, estas se hayan combinado y todos los conflictos hayan sido resueltos de la misma forma en todas las réplicas (en este ejemplo, que el valor retornado de una lectura del registro sea siempre el mismo, independientemente de la réplica), asegurando así una consistencia eventual.

Los programadores de dichos sistemas están obligados, por lo tanto, a implementar protocolos de resolución de conflictos producto de las actualizaciones de los distintos registros del sistema. Estos protocolos de resolución son llamados *replicated data types* ($RDTs$) [2]. Los mismos determinan la resolución de conflictos para todo tipo de datos, como ser registros genéricos, contadores, listas, etc. Las estrategias pueden ser tan sencillas como

ordenar a las actualizaciones por su *timestamp* y dejando ganar a la más reciente o devolver una lista de todos los valores posibles y dejar que el cliente del sistema decida cuál elegir: en el ejemplo reciente se devolverían los dos valores escritos por w_1 y w_2.

Al momento de implementar estos tipos de datos replicados, es posible demostrar que dicha implementación es correcta respecto del comportamiento especificado para los mismos en el RDT basándonos en un resultado conocido [1] que establece la existencia de una relación de simulación entre el comportamiento especificado y el comportamiento concreto de la implementación expresados como dos *labelled transition systems* ($LTSs$). Dicha demostración de existencia de una relación de simulación podría realizarse a mano o utilizando un asistente de demostraciones como es Coq.

Coq es un sistema formal de manejo de demostraciones que provee un lenguaje formal para escribir definiciones matemáticas y ejecutar algoritmos junto con un entorno para el desarrollo semi-interactivo de demostraciones de teoremas, chequeadas por computadora. A diferencia de los demostradores automáticos de teoremas, Coq incluye *tácticas* para la demostración de los mismos, que son procedimientos formales de semidecisión por las cuales se va construyendo la demostración mecánica de estos. Además, provee de un lenguaje de especificación llamado *gallina*, mediante el cual permite el desarrollo de teorías matemáticas. El lenguaje de comandos de *gallina* es llamado *Vernacular*.

Cuando Coq es visto como un sistema lógico, implementa una teoría de tipos de orden superior, mientras que cuando es visto como un lenguaje de programación, implementa un lenguaje de programación funcional tipado.

1.2. Objetivo

El objetivo de este trabajo es abordar el problema del modelado, formalización y verificación de la correcta implementación de $RDTs$ utilizando el asistente de demostraciones Coq, para así proveer de un marco de trabajo para futuras demostraciones de correctitud de implementaciones.

Para demostrar formalmente la correctitud, modelaremos en Coq dos $LTSs$: el primero, para el LTS que describe el comportamiento abstracto del RDT, es decir, el comportamiento especificado. Este define sus estados y transiciones en base a lo que la especificación permite o no; el segundo, para el LTS que describe el comportamiento concreto de una implementación del RDT, definiendo sus estados en base a los estados que toma la implementación y sus transiciones en base a las operaciones del tipo de datos, que llevan a la misma a un nuevo estado. Para modelar a estos será necesario también modelar los conceptos de *visibilidad* y *arbitración* en forma de *labelled directed aciclic graphs* ($LDAGs$) y *paths* respectivamente, como veremos en el capítulo 3. Puntualizaremos en una demostración de correctitud utilizando el tipo de datos *contador* como instancia de tipo de datos replicado y una implementación basada en estados del mismo.

1.3. Organización

El trabajo de tesis se encuentra estructurado de la siguiente manera: en el capítulo 2 introducimos las nociones y definiciones básicas que serán necesarias para el entendimiento del trabajo. También revisamos Coq y las construcciónes del mismo que serán utilizadas en

el capítulo 4. El capítulo 3 presenta el marco teórico sobre los tipos de datos replicados: la especificación de los mismos junto con los dos enfoques principales para esto, los conceptos de *visibilidad* y *arbitración*, los tipos de implementación y cómo la especificación y la implementación pueden expresarse en forma de $LTSs$. El capítulo 4 reúne los conceptos para modelar en Coq los dos $LTSs$ y muestra cómo utilizar los mismos para escribir el teorema que expresa que la implementación elegida del *Contador* es correcta respecto del comportamiento especificado. Finalmente, en el capítulo 5, hablamos sobre las conclusiones y el trabajo a futuro.

2. PRELIMINARES

En este capítulo presentamos algunas definiciones y operaciones que usaremos a lo largo de todo el trabajo. También veremos la sintaxis de Coq y cómo especificar proposiciones (*Props*) y teoremas en el mismo.

2.1. Grafos acíclicos dirigidos con etiquetas

En uno de los enfoques de especificaciones que veremos usaremos la noción de grafos acíclicos, dirigidos y etiquetados. Para ello supongamos que tenemos un conjunto de eventos E y un conjunto de etiquetas $\mathcal{L}$.

Definición 1 (*Orden parcial estricto*)**.** Un orden parcial estricto $<$ es una relación binaria que cumple que es irreflexiva y transitiva. Es decir, sea P un conjunto sobre el que se define el orden parcial estricto $<$, para todo a, b y $c \in P$ se cumplen:

$Irreflexividad$: No $a < a$.
$Transitividad$: Si $a < b$ y $b < c$, entonces $a < c$.

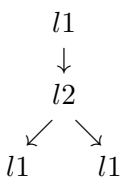
Definición 2 (*Orden total estricto*)**.** Un orden total estricto $<$ es una relación binaria que cumple que es un orden parcial estricto con la condición agregada de que dicha relación también debe ser *total*. Es decir, sea P un conjunto sobre el que se define el orden total estricto $<$, para todo a, b y $c \in P$ se cumplen:

$Irreflexividad$: No $a < a$.
$Transitividad$: Si $a < b$ y $b < c$, entonces $a < c$.
$Total$: O bien $a < b$ o bien $b < a$.

Definición 3 (*Grafo acíclico dirigido con etiquetas*)**.** Un grafo acíclico dirigido con etiquetas ($LDAG$) sobre un conjunto de etiquetas $\mathcal{L}$ es una tupla $G = \langle E_G, <_G, \lambda_G \rangle$ tal que E_G es un conjunto de eventos indicando los nodos del mismo, $<_G$ es una relación binaria $\subseteq E_G \times E_G$ tal que su clausura transitiva es un orden parcial estricto que representa las aristas del grafo y $\lambda_G : E_G \times \mathcal{L}$ es una función de etiquetado, que representa las etiquetas de los eventos en E_G.

A continuación vemos un ejemplo de un $LDAG$. Por simplicidad colocamos las etiquetas del grafo en lugar de los eventos con su función de etiquetado λ_G.

Ejemplo 1 ($LDAG$ genérico)**.**



En este caso estamos en presencia de un $LDAG$ $G = \langle E_G, <_G, \lambda_G \rangle$ definido sobre el conjunto de etiquetas $\mathcal{L} = \{l1, l2\}$, donde E_G tiene cardinalidad = 4, $<_G$ es descrita por las aristas del mismo tal que no tiene ciclos (lo que equivale a pedir que su clausura transitiva sea un orden parcial estricto) y su función de etiquetado λ_G asigna la etiqueta $l2$ al evento central y $l1$ a los demás. □

Hay una subclase particular de los $LDAGs$ que es de vital importancia para este trabajo, la clase de los $Paths$.

Definición 4 ($Path$)**.** Un $Path$ es un grafo acíclico dirigido con etiquetas $P = \langle E_P, <_P, \lambda_P \rangle$ tal que $<_P$ es un orden *total* estricto.

En otras palabras, un *path* P nos da una linealización de los eventos sobre los que se define, E_P.

Ejemplo 2 ($Path$ genérico)**.**

$$\begin{array}{ccc} & l1 & \\ \downarrow & & \searrow \\ l2 & \rightarrow & l1 \end{array}$$

Un path $P = \langle E_P, <_P, \lambda_P \rangle$ definido sobre el conjunto de etiquetas $\mathcal{L} = \{l1, l2\}$, donde E_P tiene cardinalidad = 3, $<_P$ es descrita por las aristas del mismo formando una linealización (ya que $<_P$ es un orden *total* estricto) y donde λ_P asigna las etiquetas según la figura. □

Escribimos $\mathbb{G}(\mathcal{L})$ y $\mathbb{P}(\mathcal{L})$ para referirnos al conjunto de $LDAGs$ y *paths* sobre el conjunto de etiquetas $\mathcal{L}$ respectivamente. De aquí en más, por motivos de simplicidad, en las representaciones gráficas de los paths omitiremos dibujar las aristas que pueden obtenerse por transitividad. Al mismo tiempo, para indicar que dicha simplificación se ha realizado, y no confundir entre un orden parcial y uno total, utilizaremos líneas rectas en lugar de flechas. Dicho esto, el path del ejemplo 2 lo representaremos de la siguiente forma:

$$\begin{array}{c} l1 \\ | \\ l2 \\ | \\ l1 \end{array}$$

2.2. Operaciones sobre $LDAGs$

Veamos algunas operaciones sobre grafos que será necesario tener en cuenta para los capítulos subsiguientes. A modo de simplicidad introduciremos un poco de notación.

Dado un $LDAG$ $G = \langle E, <, \lambda \rangle$, notamos como $<|_{E'}$ a la restricción de la relación $<$ a los eventos del conjunto E', es decir, $<|_{E'} = < \cap (E' \times E')$. Análogamente, definimos la restricción $\lambda|_{E'}$ para la función de etiquetado λ como la misma función de etiquetado pero restringiendo el dominio a los elementos de E'. Finalmente, notamos E_T a la extensión del conjunto E con un nuevo elemento T, es decir, $E_T = E \cup \{T\}$ donde $T \notin E$.

Definición 5 (Restricción y extensión de grafos)**.** Sea $G = \langle E, <, \lambda \rangle$ un $LDAG$ y $E' \subseteq E$, definimos:

- $G|_{E'} = \langle E', <|_{E'}, \lambda|_{E'} \rangle$ como la restricción de G a E'.
- $G^l_{E'} = \langle E_T, < \cup (E' \times \{T\}), \lambda[T \to l] \rangle$ como la extensión de G sobre los eventos E' con un nuevo evento T etiquetado con l.

Omitiremos el subíndice E' en $G^l_{E'}$ cuando $E' = E$, es decir, notaremos solo G^l cuando E' sean todos los eventos del $LDAG$ G.

Veamos algunos ejemplos sobre estas operaciones.

Ejemplo 3 (Path restringido)**.** Tomemos a P, el path genérico del ejemplo 2, y definamos $E' = \{e \mid \lambda_P(e) = l1\}$. Luego, el path restringido a E', $P|_{E'}$, tiene la siguiente representación:

$$\begin{array}{c} l1 \\ | \\ l1 \end{array}$$

□

Ejemplo 4 ($LDAG$ parcialmente extendido)**.** Tomemos a G, el grafo genérico del ejemplo 1, y definamos $E' = \{e \mid \lambda_G(e) = l2\}$. Extenderemos al mismo con un nuevo evento etiquetado con $l1$. Luego, el grafo extendido parcialmente $G^{l1}_{E'}$, resulta en el siguiente:

$$\begin{array}{ccc} & l1 & \\ & \downarrow & \\ & l2 \to l1 & \\ \swarrow & & \searrow \\ l1 & & l1 \end{array}$$

Se agregó un nuevo evento etiquetado con $l1$ y, puesto que hay un único evento en el conjunto E', ese fue el único relacionado hacia el nuevo evento $l1$. □

Ejemplo 5 ($LDAG$ totalmente extendido)**.** Tomemos nuevamente al *path* P del ejemplo 2. Buscamos extenderlo con un nuevo evento $l2$ de forma que siga siendo un *path*. Para esto, todos sus eventos deben ser relacionados hacia el nuevo evento. Como mencionamos anteriormente, puesto que $E' = E$, omitimos el subíndice E', por lo que la extensión P^{l2} queda de la siguiente forma:

$$\begin{array}{c} l1 \\ | \\ l2 \\ | \\ l1 \\ | \\ l2 \end{array}$$

Recordar también que, para los casos de *paths*, omitiremos las relaciones que puedan conseguirse por transitividad y que usaremos líneas rectas en lugar de flechas. En este ejemplo, implícitamente el primer evento estaba también relacionado al tercero (como se mencionó en el ejemplo 2) y los dos primeros eventos están también relacionados con el nuevo evento de $l2$. □

2.3. Sistema de transiciones con etiquetas (LTSs)

Será necesario también definir formalmente los sistemas de transiciones con etiquetas, puesto que serán usados luego para modelar los cambios de estados de los tipos de datos que veremos.

Definición 6 (Sistema de transiciones con etiquetas)**.** Es una tupla $\langle \mathcal{S}, \Lambda, \rightarrow \rangle$ donde $\mathcal{S}$ es un conjunto de estados (no necesariamente finito), Λ es un conjunto de etiquetas y $\rightarrow$ es una relación de transiciones etiquetadas (es un subconjunto de $\mathcal{S} \times \Lambda \times \mathcal{S}$). Escribimos a $(e, l, e') \in \rightarrow$ como $e \xrightarrow{l} e'$, la cual representa una transición desde el estado e al estado e' con etiqueta l.

2.4. Coq

Daremos una breve introducción a los aspectos de Coq que serán necesarios para entender nuestro trabajo.

2.4.1. Tipos de datos inductivos

Dado que Coq implementa un lenguaje de programación funcional, podemos entonces definir tipos de datos algebraicos. Estos tipos de datos definen reglas para la construcción de un objeto del tipo que representan. Distintas reglas deben ser consideradas como introductorias de distintos objetos. En el contexto de Coq, estos tipos de datos algebraicos son llamados tipos inductivos (*Inductive types*).

Ejemplo 6 (Tipo inductivo para booleanos)**.**

```
Inductive bool : Type :=
  | true
  | false.
```

De esta forma, `true` nos devolvería un objeto del conjunto de objetos del tipo inductivo `bool`, al igual que `false`. Lo que esta definición también dicta, es que `true` y `false` son las **únicas** formas de obtener objetos del tipo inductivo `bool`. □

Para dar un ejemplo un poco más complejo (ejemplo 7), tomemos el conjunto de todos los números naturales más el 0. Lo importante de este caso es que no se representa a un tipo de datos *enumerado* o finito, sino a un conjunto infinito, por lo que no podemos enumerar todos los posibles objetos del mismo. Para subsanar esto, tendremos que usar una forma más rica de declaración de tipos para representarlos adecuadamente.

Ejemplo 7 (Tipo inductivo para los números naturales)**.**

```
Inductive nat : Type :=
  | O
  | S (n : nat).
```

Con esta definición, 0 queda representado por `O`, 1 por `S O`, 2 por `S (S O)` y así sucesivamente. Informalmente la declaración puede leerse como sigue:

- `0` es un número natural
- `S` puede ser puesta delante de un número natural para generar así **otro** número natural. Si `n` es un número natural, entonces `S n` también lo es. □

Por último, podemos definir *aliases* de un tipo de datos definiéndolos como equivalentes, de la siguiente forma:

```
Definition boolean : Type := bool.
```

2.4.2. Funciones

En Coq podemos definir funciones, es decir, procedimientos que toman cero, uno o más argumentos y devuelven un valor. Dado que el lenguaje definido por Coq es un lenguaje funcional, las funciones en Coq deben ser funciones matemáticas, es decir, deben ser totales (para todo argumento existe un valor de retorno) y deben respetar la unicidad (hay uno único valor de retorno para una secuencia de argumentos provistos).

Tenemos dos tipos de funciones principales sobre Coq, las que hacen uso de la recursión y las que no. Las primeras deben declararse usando la palabra reservada **`Fixpoint`** mientras que las últimas son declaradas usando **`Definition`**.

Ejemplo 8 (Función recursiva para Fibonacci)**.**

```
Fixpoint fibonacci (n: nat) : nat :=
   match n with
      | 0 => 0
      | S 0 => S 0
      | S n' => (fibonacci n') + (fibonacci (n' - 1))
   end.
```

Esta función recibe un parámetro `n` de tipo `nat` y devuelve un elemento también de tipo `nat` que representa el valor correspondiente al argumento para la secuencia de Fibonacci. La sentencia `match n with ... end` nos permite hacer *pattern matching* sobre `n`. □

Veamos ahora un ejemplo sobre una función sencilla no recursiva.

Ejemplo 9 (Igualdad de dos booleanos)**.**

```
Definition eqb (b1 b2: bool) : bool :=
   match b1, b2 with
      | true, true => true
      | false, false => true
      | _, _ => false
   end.
```

Esta función recibe dos parámetros `b1`, `b2` de tipo `bool` y devuelve un elemento de tipo `bool` cuyo valor de verdad representa si `b1` y `b2` son iguales. La escritura `_` se usa para indicar que no importa que valor tomen `b1` y `b2`. □

2.4.3. Props y teoremas

Las proposiciones (`Props`) en Coq definen un conjunto infinito de elementos que tienen la capacidad de que su veracidad sea (o no) demostrada formalmente. Estas proposiciones son las que luego serán usadas para especificar y demostrar teoremas.

Como ya hemos mencionado, Coq no es un demostrador automático de teoremas, por lo que Coq no puede decirnos si la veracidad de una proposición es efectiva o no de forma automática, pero nos provee de tácticas que nos ayudan a simplificar y transformar las proposiciones para tratar de demostrar que una proposición es en realidad equivalente a la proposición que representa la veracidad universal, la proposición `True`. Si se logra demostrar dicha equivalencia, entonces la proposición es válida. De lo contrario, si no podemos lograr mostrar dicha equivalencia (y si no se alteran los axiomas iniciales de Coq) entonces puede suceder que la proposición no sea verdadera, o que tengamos que buscar otra posible transformación y/o simplificación de la misma para lograrlo.

Así como existe la proposición trivial `True`, la cual es trivialmente demostrable puesto que ya es equivalente a sí misma, existe la proposición trivial `False`, la cual no es demostrable por ninguna vía de transformación o simplificación puesto que representa lo contrario.

Veamos algunos ejemplos de `Props`:

Ejemplo 10 (`Props` usando cuantificadores universales y existenciales)**.**

- `forall (n m: nat), n = m`
- `forall (n: nat), n=n -> True`
- `exists (n: nat), n*10 = 1000`
- `exists (n: nat), False`

En el primer caso tenemos una `Prop` que declara que para todo par de naturales `n` y `m` vale que `n = m`. En el segundo caso se muestra una `Prop` que indica que, para todo natural `n`, si vale `n=n`, entonces vale la proposición trivial `True`. Los últimos dos casos serán demostrables si existe al menos un natural `n` que cumpla lo que cada expresión dicta. Es claro que solo la primera de ambas será demostrable. □

Con respecto a los teoremas, podemos pensar a los mismos en Coq como proposiciones nombradas, a las cuales se las demuestra por válidas. Como ya hemos mencionado, para efectuar la demostración de un teorema se usan las llamadas *tácticas* que Coq nos provee. Cuando estamos demostrando un teorema, el *contexto* es el conjunto de proposiciones asumidas (hipótesis) que tenemos en un momento dado. En cualquier momento de la demostración, la proposición que estamos intentamos demostrar cierta es llamada: el *goal*.

Un teorema también puede servirnos para demostrar otros teoremas (generalmente a través de las tácticas *apply* o *rewrite*).

Para declarar un teorema en Coq se utiliza la keyword `Theorem` y para dar inicio y fin a una demostración se utilizan `Proof` y `Qed`.

Ejemplo 11 (La igualdad se preserva frente al mismo desfasaje)**.**

```
Theorem same_offset_preserves_equality : forall (n m o: nat), n=m -> n+o=m+o.
   Proof.
      intros n m o.
      intros equality_n_m.
      rewrite equality_n_m.
      reflexivity.
   Qed.
```

Dado que el teorema predica sobre tres variables universales `n m` y `o`, debemos asumirlas primero en nuestro contexto, para ello utilizamos la táctica `intros`. También agregamos la hipótesis `n=m` al contexto y la llamamos equality_n_m. Con la táctica `rewrite equality_n_m` reemplazamos cualquier aparición de `n` en el *goal* por su igual `m`. Finalmente `reflexivity` termina la demostración puesto que ambos lados de la igualdad son idénticos. □

Puede consultarse un listado completo de todas las tácticas disponibles en Coq en [3].

2.4.4. Proposiciones inductivamente definidas

Otra forma de definir proposiciones (*Props*) es de manera inductiva. En este caso lo que hacemos es definir las posibles formas o reglas mediante las cuales se cumple que una proposición definida por el usuario es cierta.

Supongamos que necesitamos tener una proposición que nos diga si una variable `n: nat` es par (ejemplo tomado de [4]). Supongamos también que disponemos de una función que toma un `nat` y devuelve un `bool` indicando si dicho natural es par, llamémosla `isEven`.

Una posible implementación de nuestra *Prop* sería: `isEven n = true`, puesto que como `isEven` retorna un `bool` debemos compararlo contra `true` para que sea una `Prop`.

Otra posibilidad, que no requiere la intervención de booleanos, es decir que `n` es par si podemos establecer su paridad a partir de las siguientes dos reglas:

$$\frac{}{even\ 0}\ \texttt{(even\_0)} \qquad\qquad \frac{even\ n}{even\ S\ (S\ n)}\ \texttt{(even\_SS)}$$

- Regla `even_0` : El número 0 es siempre par
- Regla `even_SS`: Si `n` es par, entonces `S (S n)` es también par.

Una forma de expresar esta proposición en Coq de manera inductiva es la siguiente:

```
Inductive even : nat -> Prop :=
   | ev_0 :
```

```
        even 0
| ev_SS (n: nat) :
    even n ->
        even S (S n).
```

Declaramos dos reglas, una de las cuales no tiene una hipótesis (la regla `ev_0`) y una que si, que indica que para que `S (S n)` sea `even`, `n` debe ser `even` primero. Existen otras formas de declarar dicha hipótesis, pero esta es la que elegimos para este trabajo.

En este trabajo utilizamos las proposiciones inductivamente definidas para representar las transiciones válidas de distintos $LTSs$, en particular: los $LTSs$ abstractos y concretos, de los cuales hablaremos en los capítulos posteriores.

Para terminar, veamos un ejemplo de una proposición inductivamente definida no recursiva (no hay hipótesis sobre la misma proposición como en el caso de `even`).

Ejemplo 12 (Transiciones válidas del estado de un carrito de compras)**.** Modelaremos las transiciones válidas que puede tener un carrito de compras. Las transiciones ocurren cuando se agregan o se remueven elementos del mismo. Para modelar dichas operaciones usaremos el siguiente tipo inductivo, donde `e` es el tipo del producto agregado al carrito o removido del mismo (puede haber más de uno del mismo tipo).

```
Inductive op : Type :=
   | agregar (e: nat)
   | remover (e: nat).
```

Y las reglas que determinan cuáles son transiciones válidas del estado del carrito, al cual modelaremos como una lista de naturales (`list nat`) con los productos que actualmente tiene.

```
Inductive transiciónCarrito : list nat -> op -> list nat -> Prop :=
   | elementoAgregado (l l': list nat) (e: nat) :
      cons e l = l' ->
         transiciónCarrito l (agregar e) l'.
   | elementoRemovido (l l': list nat) (e: nat) :
      removerUno e l = l' ->
         transiciónCarrito l (remover e) l'.
```

Para el caso del elemento agregado, la transición `transiciónCarrito l (agregar e) l'` solo será válida si se cumple que `l'` es el resultado de agregar `e` a `l` (`cons` es el constructor de listas que agrega un elemento a una lista ya existente). Para el caso del elemento removido, la transición `transiciónCarrito l (remover e) l'` solo será válida si `l'` es el resultado de eliminar uno de los elementos iguales a `e` de la lista `l` actual. Por simplicidad del ejemplo asumimos que tenemos una función `removerUno` que opera sobre una lista eliminando solo el primer elemento igual a `e` que encuentra.

Podría acusarse a este ejemplo de sobreespecificar el comportamiento ya que determina un orden sobre `l` y porque selecciona unívocamente cuál debe ser el elemento removido de

entre todos los posibles, pero a efectos del ejemplo esto no es un problema. □

3. RDTS

Para especificar los *RDTs*, la literatura propone dos enfoques, el enfoque tradicional, el cual se caracteriza por determinar el resultado de una operación dado cierto contexto, y el enfoque funcional, el cual es una forma más general del tradicional y se caracteriza por determinar todas las computaciones que serían válidas según el contexto actual. Este último enfoque da vida también a una percepción del mismo en forma de *labelled trasition system* (LTS), donde una transición se genera cada vez que un nuevo evento, sobre el objeto que el RDT modela, ocurre.

Las implementaciones de los *RDTs* pueden ser de dos tipos principales, los cuales son las implementaciones basadas en operaciones y las basadas en estados. Describiremos aquí también las características de cada una junto con ejemplos concretos de implementaciones para ambas.

3.1. Especificación

Para una correcta comprensión sobre la especificación de los tipos de datos replicados, es necesario familiarizarse primero con los conceptos de *visibilidad* y *arbitración*. Los mismos determinan relaciones entre las operaciones que ocurran en un objeto replicado que serán una herramienta útil a la hora de especificar los *RDTs*. La *visibilidad* determina, dada una operación, cuáles de las operaciones previas a esta son conocidas por la misma, lo cual podría influir en el valor retornado por dicha operación, mientras que la *arbitración* determina cómo el sistema ordena las operaciones para desambiguar la importancia o precedencia que dos operaciones tienen entre sí relativamente para poder seleccionar una en caso de necesitarlo.

Dado un conjunto de eventos E, donde cada evento representa una operación realizada sobre el objeto que se está modelando, la *visibilidad* indica el conocimiento que cada evento tiene de los otros eventos en E, es decir, de qué operaciones es consciente cada operación. En el contexto de un sistema centralizado, cada operación tendría conocimiento de todas las operaciones previas y carecería de sentido hablar sobre qué eventos cada evento conoce, pero en un entorno replicado esto no es así y aparece la necesidad de modelar dicho conocimiento. Una restricción importante de la *visibilidad* es que dos eventos no pueden conocerse mutuamente. Esto tiene sentido puesto que siempre uno de los dos se manifiesta primero, imposibilitando así que pueda tener conocimiento de un evento que aún no ha ocurrido.

Veámoslo con un ejemplo que nos dará una mayor intuición de cómo son utilizados los conceptos de *visibilidad* y *arbitración*:

Ejemplo 13. Supongamos que tenemos un registro genérico, `intereg`, que sólo tiene operaciones de lectura rd y escritura $w(a),\ a \in \mathbb{N}$. Si suponemos que en cierto punto sólo se han realizado dos operaciones de escritura, $w(x)$ y $w(y)$, y luego una operación de lectura rd inmediatamente posterior, el valor de la operación de lectura dependerá de la visibilidad que esta operación tenga. Si rd sólo tiene visibilidad sobre $w(x)$ el valor de

retorno será x, mientras que será y en caso de que sólo tenga visibilidad de $w(y)$. Así mismo, si rd no tiene visibilidad sobre ninguna de las dos escrituras, devolverá el valor por default asociado al registro (0 por ejemplo). Existe un último caso, en el cual rd tiene visibilidad sobre ambas escrituras, en donde podría devolverse un conjunto con todos los posibles valores, o elegir uno de ellos. En este último escenario el concepto de *arbitración* entra en juego. □

La *arbitración* sobre un conjunto de eventos E define un orden estricto y total sobre los mismos, es decir, una linealización de estos. Puede pensarse a la misma como indicadora de la importancia, prioridad o precedencia que hay entre cada par de eventos. Esta herramienta es utilizada en la especificación de los $RDTs$ cuando se necesita desambiguar un comportamiento.

En el ejemplo 13 anterior, frente al caso en que rd sea consciente de ambas escrituras, se utilizaría la *arbitración* para "*desempatar*", tomando como respuesta el valor escrito por la escritura más "importante". Como instancia de una posible arbitración podríamos elegir el momento en que ambas escrituras ocurrieron, siendo siempre la escritura más reciente la más importante. Esta, aunque parezca la arbitración más lógica, no es la única, podríamos definir cualquier orden que necesitemos para nuestro caso particular.

Finalmente, definimos formalmente la *visibilidad* y la *arbitración*:

Definición 7 (*Visibilidad sobre un conjunto de eventos*)**.** Se define la visibilidad sobre un conjunto de eventos E como una relación en $E \times E$ acíclica.

Definición 8 (*Arbitración sobre un conjunto de eventos*)**.** Se define la arbitración sobre un conjunto de eventos E como una relación transitiva, irreflexiva y total en $E \times E$.

Procedamos ahora a conocer los dos enfoques de especificación sobre $RDTs$ que existen.

3.1.1. Enfoque tradicional

En pos de definir la especificación de un tipo de datos replicado según el enfoque tradicional [2] es necesario primero definir el concepto de *operation context* que esta usa.

Definición 9 (*Operation context*)**.** Se define el ***operation context*** de un tipo de datos τ como una tupla $L = (o, E, op, vis, ar)$ donde:

- o pertenece al conjunto de operaciones asociadas al tipo de datos τ, el cual llamaremos Op_τ.
- E es el conjunto de todos los eventos de los que o tiene conocimiento. Es un subconjunto finito de $Event$, el conjunto de todos los eventos posibles.
- op es una función: $Event \to Op_\tau$ que indica la operación asociada a cada evento.
- vis es la visibilidad sobre E.
- ar es la arbitración sobre E.

Definido el *operation context*, la especificación de un tipo de datos replicado en el esquema de especificación tradicional queda bien definida:

Definición 10 (*Especificación en el enfoque tradicional*)**.** La especificación de un tipo de datos replicado τ se define como una función F_τ tal que, dado un *operation context* L para τ, especifica un valor de retorno $F_\tau(L) \in Val_\tau$, siendo Val_τ el conjunto de valores asociado al tipo τ.

Ejemplo 14 (*Contador*)**.** Un *contador* sólo tiene operaciones de lectura (*read*) e incremento (*inc*) de su valor, iniciándose el mismo en el valor 0. Un *read* debe retornar siempre un valor igual a la cantidad de *incs* de los que tiene conocimiento, mientras que un *inc* no tiene valor de retorno. Queda entonces definida su especificación como:

$$F_{ctr}(inc, E, op, vis, ar) = \bot$$
$$F_{ctr}(read, E, op, vis, ar) = |\{e \in E | op(e) = inc\}|$$

Notar que en este caso no se utiliza la visibilidad vis al momento de la operación de *read*. Esto es debido a que, por definición, E es el conjunto de todos los elementos ya visibles por esta.

Tampoco se utiliza la arbitración ar, debido a que en un *contador* las operaciones de incremento son conmutativas, es decir, el resultado de una lectura luego de una cierta cantidad de operaciones de incremento es el mismo para todos los posibles órdenes en que estos hayan sucedido. □

Veamos la aplicación de la arbitración ahora para el ejemplo 13 antes presentado, eligiendo como arbitración la estrategia *last writter wins* (LWW), donde el último write es el más prioritario.

Ejemplo 15 (intreg con LWW)**.** Como vimos, las posibles operaciones son de lectura (rd) y de escritura ($w(a),\ a \in \mathbb{N}$).

$$F_{intreg}(wr(a), E, op, vis, ar) = \bot$$

$$F_{intreg}(rd, E, op, vis, ar) = \begin{cases} a & \text{si } E^{ar} = \xi_1 wr(a) \xi_2 \text{ y } \nexists\ wr(b) \in \xi_2 \\ 0 & \text{sino} \end{cases}$$

donde E^{ar} denota los eventos de E ordenados según la relación de arbitración ar y $\xi_1,\ \xi_2$ son secuencias de operaciones. De esta forma, la operación de escritura no retorna ningún valor, mientras que la operación de lectura opera retornando el último write (en términos de la relación ar) si existe, y 0 sino. Nuevamente, vis no se utiliza puesto que, por definición, E son todos los eventos de los que la operación de rd tiene conocimiento, y no nos interesan las visibilidades internas que haya entre los nodos de E. □

Veamos un último ejemplo donde la arbitración no es necesaria, pero sí es necesario revisar las visibilidades internas entre los nodos de E. Se trata de un registro multivalor (`mvr`), en el cual una lectura rd retorna todos los posibles resultados en lugar de priorizar solo el valor escrito por algún wr.

Ejemplo 16 (Registro multivalor `mvr`)**.**

$$F_{mvr}(wr(a), E, op, vis, ar) = \bot$$

$$F_{mvr}(rd, E, op, vis, ar) = \{a \mid \exists e \in E, op(e) = wr(a) \wedge \nexists\ e' \in E, op(e') = wr(\_) \wedge e \xrightarrow{vis} e'\}$$

La operación de escritura, como antes, devuelve $\perp$. El caso interesante aparece en el rd. A priori rd devolverá un conjunto con todas las aes que cumplan que existe un write que las haya escrito, pero esto no es suficiente puesto que la idea del registro `mvr` es devolver todos los valores posibles que podría tener el registro actualmente, no todos los valores escritos históricamente, por lo que deben quitarse aquellos $wr(a)$ que fueron sobreescritos posteriormente. Esto se realiza en la segunda parte de la conjunción, donde lo que se pide es que no exista otro $wr(\_)$ (con $\_$ simbolizamos que no importa el valor escrito) tal que, teniendo visibilidad sobre el $wr(a)$, lo haya sobreescrito.

Como instancia del ejemplo, podemos tomar el siguiente escenario de visibilidades, donde el valor esperado por una lectura rd es el conjunto $\{2, 3\}$:

$$\begin{array}{ccc} & wr(0) & \\ & \downarrow & \\ & wr(1) & \\ \swarrow & & \searrow \\ wr(2) & & wr(3) \end{array}$$

□

Como fue mencionado en [1] podemos caracterizar a estas especificaciones de una forma equivalente, en términos de *Labelled Directed Acyclic Graphs* (*LDAGS*). De esta forma el *operation context* queda definido en términos de un par $\langle G, P\rangle$, con G un $LDAG$ y P un *path* sobre los eventos de G con idéntica función de labelling, es decir, $P \in \mathbb{P}(E_G, \lambda_G)$. G representa la visibilidad (vis) mientras que P la arbitración (ar). Si escribimos $\mathbb{C}(L)$ para referirnos al conjunto de *contexts* sobre el lenguaje de etiquetado L, y fijamos un set O y V de operaciones y valores respectivamente, podemos dar la definición equivalente a continuación:

Definición 11. Una especificación en el enfoque tradicional sobre un tipo de datos replicado τ es una funcion $F_\tau : O \times \mathbb{C}(O) \to V$.

Ejemplo 17 (*Contador* en términos de $LDAGs$)**.**

$$F_{ctr}(inc, \langle G, P\rangle) = \perp$$
$$F_{ctr}(read, \langle G, P\rangle) = |\{e \in E_G | \lambda(e) = inc\}|$$

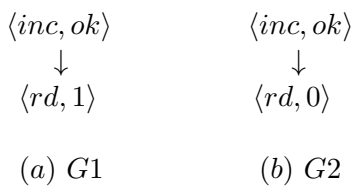
□

3.1.2. Enfoque funcional

En el enfoque funcional, propuesto en [1], la especificación denota una vista más abstracta de los RDTs. En esta, dado un $LDAG$ de visibilidad G, se devuelven todos los posibles *paths* P que representan arbitraciones admisibles (válidas) para el RDT, para los eventos ocurridos.

Definición 12 (*Especificación en el enfoque funcional*)**.** Una especificación $\mathcal{S}$ se define como una función $\mathcal{S} : \mathbb{G}(L) \to 2^{\mathbb{P}(L)}$ tal que $\mathcal{S}(\epsilon) = \{\epsilon\}$ y $\forall G,\ \mathcal{S}(G) \in 2^{\mathbb{P}(E_G, \lambda_G)}$.

Notar que, dado un $P \in \mathcal{S}(G)$, P está definido sobre los mismos eventos de G, con misma función de etiquetado, por lo que P dicta un orden total sobre todos los eventos de G. El



(a) $G1$ (b) $G2$

Fig. 3.1: Dos *LDAGs* para el RDT *contador*

hecho de que una especificación mapee al conjunto vacío para un grafo dado G significa que no existe ninguna arbitración válida de los eventos de G para el RDT.

Ejemplo 18 (*Contador* en el enfoque funcional)**.** Llamemos L al conjunto de etiquetas del *contador*. Dado que el mismo solo tiene operaciones *read* e *inc*, con valores de retorno $\mathbb{N}_0$ y *ok* respectivamente, L será $\{\langle inc, ok\rangle\} \cup (\{rd\} \times \mathbb{N}_0)$. La especificación queda definida entonces como:

$$P \in \mathcal{S}_{ctr}(G) \textit{ sii}$$
$$\forall e \in E_G, \forall k, \lambda(e) = \langle rd, k\rangle \textit{ implica que } k = \#\{e' | e' <_G e \textit{ y } \lambda_G(e') = \langle inc, ok\rangle\}$$

Notar que en este RDT, no se exige ningún orden particular sobre los eventos en P (esto es análogo a la no utilización de la relación ar para el *contador* en el caso del enfoque tradicional del ejemplo 14), por lo que cualquier *path* que tenga los mismos eventos y misma función de etiquetado que G pertenecerá a $\mathcal{S}_{ctr}(G)$, siempre que la condición expresada sobre G en la definición de la especificación se cumpla.

Analizemos los *LDAGs* de la figura 3.1. Para $G1$, vemos que existe un único elemento e etiquetado con $\langle rd, 1\rangle$ y que $1 = \#\{e' | e' <_{G1} e \textit{ y } \lambda_{G1}(e') = \langle inc, ok\rangle\}$, por lo que la condición se cumple. Luego cualquier *path* P que tenga los mismos eventos (y etiquetados idénticamente) que $G1$ será tal que $P \in \mathcal{S}_{ctr}(G1)$. En cuanto a $G2$, vemos que existe un evento e etiquetado con $\langle rd, 0\rangle$ pero $0 \neq \#\{e' | e' <_{G2} e \textit{ y } \lambda_{G2}(e') = \langle inc, ok\rangle\} = 1$. Dado que esta condición es siempre falsa, ningún *path* P será tal que $P \in \mathcal{S}_{ctr}(G2)$, por lo que $\mathcal{S}_{ctr}(G2) = \varnothing$. □

Al igual que como hicimos en el enfoque tradicional (ejemplo 15), veamos la implementación de un `intreg` con la política *last writter wins* (LWW) en el enfoque funcional.

Ejemplo 19 (Intreg con LWW en el enfoque funcional)**.** Asumamos por simplicidad que el valor de retorno de una lectura de un `intreg` aún no escrito es $\perp$. Dado que el mismo sólo tiene operaciones rd y $wr(a)$, $a \in \mathbb{N}$, con valores de retorno $\mathbb{N} \cup \{\perp\}$ y *ok* respectivamente, L será $\{\langle wr(a), ok\rangle \mid a \in \mathbb{N}\} \cup (\{rd\} \times \mathbb{N} \cup \{\perp\})$. La especificación queda definida entonces como:

$$P \in \mathcal{S}_{lwwR}(G) \textit{ sii}$$

$$\forall e \in E_G, \begin{cases} \lambda(e) = \langle rd, \perp\rangle \textit{ implica que } \forall e' <_G e, \forall k, \lambda(e') \neq \langle wr(k), ok\rangle \\ \forall k, \lambda(e) = \langle rd, k\rangle \textit{ implica que } \exists e' <_G e, \lambda(e') = \langle wr(k), ok\rangle \textit{ y} \\ \qquad \forall e'' <_G e, e' <_P e'' \textit{ implica que } \forall k', \lambda(e'') \neq \langle wr(k'), ok\rangle \end{cases}$$

La primer condición denota que un rd retorna el valor $\perp$ cuando no tiene visibilidad de ningún wr. En la segunda condición se especifica que un rd retorna un valor k cuando existe una operación e' de escritura $wr(k)$ de la cual el rd tiene visibilidad y la arbitración P posiciona a e' como el más prioritario de entre todas las escrituras e'' de las cuales rd tiene visibilidad. En otras palabras, un $LDAG$ G tiene arbitraciones admisibles sólo cuando cada evento asociado con un rd retorna el valor previamente escrito. □

3.1.3. Equivalencia

Como se demostró en [1], existe una relación entre los RDTs del enfoque tradicional y las especificaciones del enfoque funcional. En particular, las especificaciones del enfoque funcional son una forma más general de modelado respecto de los RDTs descriptos en [2], es decir, los RDTs son una clase particular de las especificaciones, y, por lo tanto, proveen una completa caracterización abstracta de los RDTs.

En este trabajo nos basamos en el enfoque de especificación funcional.

3.1.4. Especificaciones e implementaciones como labelled transition systems (LTSs)

Como se vió en [1], las especificaciones del enfoque funcional naturalmente denotan un sistema de transiciones con etiquetas, donde las transiciones se producen al momento en que un nuevo evento ocurre y los estados válidos del sistema son aquellas configuraciones que son computaciones admisibles para la especificación en cuestión. A dichos $LTSs$ los llamamos *abstract* $LTSs$. Por simplicidad, repetimos a continuación la definición sobre los *abstract* $LTSs$ presente en [1].

Definición 13 (*One-replica abstract LTS*)**.** Sea $\mathcal{S}$ una especificación, se define el abstract one-replica LTS $\mathcal{T}_\mathcal{S}$ como un sistema de transiciones con etiquetas tal que $\langle G, P\rangle$ son sus estados y $\langle op, v\rangle$ las etiquetas de sus transiciones, tales que:

- $\langle G, P\rangle$ es un estado si y solo si $P \in \mathcal{S}(G)$
- $\langle G, P\rangle \xrightarrow{\ell} \langle G', P'\rangle$ es una transición siempre que $G' = G^\ell$ y $P'|_{E_G} = P$ para toda etiqueta ℓ.

Dado que los estados $\langle G, P\rangle$ de $\mathcal{T}_\mathcal{S}$ son tales que $P \in \mathcal{S}(G)$, entonces los mismos representan sólo computaciones admisibles para $\mathcal{S}$. Luego, dado que P' preserva el orden relativo de los elementos de P y que G' es una extensión de G frente a una transición etiquetada con ℓ, dicha transición solo será posible si $\mathcal{S}$ permite dicha extensión (no puede ocurrir que dicha transición nos lleve a un estado inválido para $\mathcal{S}$).

Ejemplo 20 (Transiciones del LTS abstracto de un *contador*, $\mathcal{T}_{ctr}$)**.** Veamos a continuación las posibles transiciones del LTS abstracto de un *contador* en un contexto donde se han hecho sólo un incremento y una lectura, con un *path* ordenado en orden contrario. Partiremos del siguiente estado que, como veremos, es válido para $\mathcal{T}_{ctr}$.

$$\left(\begin{array}{c} \langle inc, ok\rangle \\ \downarrow \\ \langle read, 1\rangle \end{array} \quad , \quad \begin{array}{c} \langle read, 1\rangle \\ \downarrow \\ \langle inc, ok\rangle \end{array}\right)$$

Como se puede ver, dado que el único *read* del grafo de visibilidad G (izquierda) tiene como valor de retorno la misma cantidad de *incs* de las que tiene conocimiento, entonces cumple con la especificación del *contador* y admite una o más arbitraciones para sus eventos, una de las cuales es el path P asociado a la derecha. Finalmente, dado que los eventos de G y P son los mismos (única condición sobre P para que $P \in \mathcal{S}_{ctr}(G)$), entonces $\langle G, P\rangle$ es un estado válido del LTS, tal como indica la primer condición de la definición 13.

Veamos ahora las posibles transiciones que ocurren frente a un nuevo evento de $\langle read, 1\rangle$.

$$\left(\begin{array}{c}\langle inc, ok\rangle \\ \downarrow \\ \langle read, 1\rangle\end{array}, \begin{array}{c}\langle read, 1\rangle \\ \downarrow \\ \langle inc, ok\rangle\end{array}\right)$$

$$\langle read, 1\rangle \swarrow \qquad \downarrow \langle read, 1\rangle \qquad \searrow \langle read, 1\rangle$$

$$\left(G^{\langle read,1\rangle}, \begin{array}{c}\langle read, 1\rangle \\ \downarrow \\ \langle inc, ok\rangle \\ \downarrow \\ \langle read, 1\rangle\end{array}\right) \left(G^{\langle read,1\rangle}, \begin{array}{c}\langle read, 1\rangle \\ \downarrow \\ \langle read, 1\rangle \\ \downarrow \\ \langle inc, ok\rangle\end{array}\right) \left(G^{\langle read,1\rangle}, \begin{array}{c}\langle read, 1\rangle \\ \downarrow \\ \langle read, 1\rangle \\ \downarrow \\ \langle inc, ok\rangle\end{array}\right)$$

Donde $G^{\langle read,1\rangle}$ es:

$$\begin{array}{c}\langle inc, ok\rangle \\ \downarrow \quad \searrow \\ \langle read, 1\rangle \rightarrow \langle read, 1\rangle\end{array}$$

Lo que ha ocurrido es que, frente a un nuevo evento $\langle read, 1\rangle$, el grafo de visibilidad G se extendió desde todos sus nodos a este nuevo evento. Esto da lugar a 3 nuevas arbitraciones válidas, que son las que podemos apreciar en cada uno de los nuevos estados a los que el LTS transiciona (notar que los últimos 2 estados parecen iguales pero en realidad son las dos posibles permutaciones de los dos eventos distintos de $\langle read, 1\rangle$). Todas estas abritraciones P' cumplen que si las restringimos a los eventos de G nos da como resultado P, por lo que se cumplen todas las condiciones de la definición 13 para que las transiciones sean válidas. Como todos los nuevos P' comparten los mismos nodos que $G^{\langle read,1\rangle}$ y $\mathcal{S}_{ctr}(G) \neq \varnothing$, es claro que todos los nuevos estados son también estados válidos (cumplen la primer regla de la definición 13).

Intuitivamente, lo que $\mathcal{T}_{ctr}$ nos indica, es que frente a una nueva operación de $\langle read, 1\rangle$ que conoce todas las operaciones previas, debe cumplirse que haya exactamente 1 operación de $\langle inc, ok\rangle$ para que sea una computación válida, lo cual permitirá una o más arbitraciones posibles, en las cuales el orden de los eventos sólo debe cumplir la restricción de preservar el orden relativo de la arbitración asociada al estado desde el que se transicionó.

En caso de que dicho *read* no hubiera tenido conocimiento de exactamente un $\langle inc, ok\rangle$, lo que hubiera ocurrido es que $\mathcal{S}_{ctr}(G^{\langle read,1\rangle}) = \varnothing$, por lo que no podría existir una arbitración que admita dicho comportamiento, es decir, $\nexists P' \in \mathcal{S}_{ctr}(G^{\langle read,1\rangle})$. Luego, no podría existir un estado $\langle G^{\langle read,1\rangle}, P'\rangle$ al cual transicionar puesto que no sería un estado válido (no cumpliría la primera regla de la definición 13). □

La implementación de un tipo de datos replicado también puede modelarse con un LTS, al cual llamaremos *concrete LTS*, que representa cómo varía el estado actual de cada réplica a partir de las operaciones que en esta se realizan. Más formalmente:

Definición 14 (*Concrete LTS*)**.** Sea Σ el conjunto de posibles estados de la implementación de un RDT para una especificación $\mathcal{S}$. Un *concrete LTS*, $\mathcal{C}_\mathcal{S}$, es un sistema de transiciones etiquetadas que tiene como conjunto de etiquetas a $\mathcal{A}_\mathcal{S} = \mathcal{L} \cup (\{send, rcv\} \times \Sigma) \cup \{\tau\}$, siendo $\mathcal{L} = \mathcal{O} \times \mathcal{V}$ y τ representando transiciones internas de la implementación. Las etiquetas $\langle send, \sigma \rangle$ y $\langle rcv, \sigma \rangle$ son operaciones especiales usadas por las réplicas para sincronizarse entre sí. Los estados de $\mathcal{C}_\mathcal{S}$ se corresponderán con los posibles estados Σ que podrá tener la implementación.

Una implementación completa de un RDT se obtiene luego poniendo varias réplicas en paralelo, todas ellas respetando el mismo *concrete LTS*. Veamos a continuación un ejemplo para una implementación basada en estados del *contador*.

Ejemplo 21 (Implementación para el tipo de datos *Contador*)**.** En pos de denotar la implementación del RDT debemos definir Σ, $\mathcal{L}$ y las transiciones del *concrete LTS* asociado.

- $\Sigma = \mathcal{R} \times (\mathcal{R} \mapsto \mathbb{N}_0)$, donde $\mathcal{R}$ denota el conjunto de identificadores de las réplicas. La primer componente de estos pares representa la réplica del estado, mientras que la segunda representa una función que indica cuántos incrementos realizó cada réplica. Utilizaremos la notación $\langle \bullet, \bullet \rangle$ para referirnos a las tuplas pertenecientes a Σ. De esta forma, $\langle r, v \rangle$ representa un elemento de Σ, donde r es una réplica id y v es una función tal que $v(r_i)$ indica la cantidad de incrementos hechos por la réplica r_i.
- $\mathcal{L} = \{\langle inc, ok \rangle\} \cup (\{rd\} \times \mathbb{N}_0)$ denota el conjunto de operaciones del tipo de datos.
- Las transiciones del *concrete LTS* asociado quedan determinadas por las reglas de inferencia a continuación:

$$\frac{k = \sum_{s \in \mathcal{R}} v(s)}{\langle r, v \rangle \xrightarrow[rd,k]{} \langle r, v \rangle}\ \texttt{(READ)} \qquad \langle r, v \rangle \xrightarrow[inc,ok]{} \langle r, v[r \mapsto v(r) + 1] \rangle\ \texttt{(INC)}$$

$$\langle r, v \rangle \xrightarrow[send,\langle r,v \rangle]{} \langle r, v \rangle\ \texttt{(SEND)} \qquad \langle r, v \rangle \xrightarrow[rcv,\langle r',v' \rangle]{} \langle r, max\{v, v'\} \rangle\ \texttt{(RCV)}$$

donde $max\{v, v'\}$ representa una función tal que para todo valor $r \in \mathcal{R}$, $max\{v, v'\}(r) = max(v(r), v'(r))$.

La regla de read nos indica que sólo podremos transicionar por la etiqueta rd, k si los estados origen y destino son el mismo, y si k es igual a la suma de todos los incrementos hechos por todas las réplicas. Para esto, sumamos todos los valores asociados a todos los puntos del dominio de v, la función asociada al estado. La regla de inc nos indica que siempre podremos transicionar por la etiqueta inc, ok y el estado al que llegaremos es exactamente el mismo estado pero con el valor asociado a la réplica r en v incrementado en uno, ya que es esta la que recibió la operación

de incremento. La transición por send no tiene restricciones, se transiciona hacia el mismo estado a partir de hacer un send de este. Finalmente, la regla de rcv nos indica que, al recibir otro estado, nuestro estado se modificará para contabilizar siempre la máxima cantidad de incrementos hechos por cada réplica.

Notar que en esta implementación, se carece de transiciones τ. □

3.2. Implementación: estados vs operaciones

Las implementaciones de los tipos de datos replicados pueden dividirse en dos categorias principales: implementaciones basadas en **estados** e implementaciones basadas en **operaciones**. Formalmente, una implementación de un RDT se define como sigue ([2]).

Definición 15 (*Implementación de un RDT*)**.** Una implementación para un *tipo de datos replicado* τ es una tupla $\mathcal{D}_\tau = (\Sigma, \vec{\sigma_0}, M, do, send, receive)$, donde Σ es el conjunto de todos los estados posibles por los que puede transicionar la implementación, $\vec{\sigma_0} : ReplicaID \to \Sigma$, M representa el conjunto de mensajes que pueden comunicarse entre las réplicas, $do : Op_\tau \times \Sigma \times Timestamp \to \Sigma \times Val_\tau$, $send : \Sigma \to \Sigma \times M$ y $receive : \Sigma \times M \to \Sigma$.

$\vec{\sigma_0}$ es una función que representa el estado inicial de cada réplica. do, dada una operación $\in Op_\tau$, un estado Σ y un $Timestamp$, indica qué valor $\in Val_\tau$ dicha operación retorna y cómo se modifica el estado de la implementación frente a dicha operación.

Finalmente *send* y *receive* son las operaciones que producen y reciben mensajes respectivamente que serán o fueron broadcasteados a todas las réplicas. *receive* retorna el nuevo estado al que la implementación transiciona luego de fusionar el estado actual con el recibido. *send* opcionalmente también podría alterar el estado actual de la implementación como se verá en el ejemplo del *contador* basado en operaciones a continuación, es por esto que además de retornar el mensaje a enviar retorna un estado.

Las implementaciones deben asegurar la convergencia de estados en todas las réplicas, es decir, si en algún momento todas las réplicas han comunicado todas las operaciones de las que tienen conocimiento, y estas han sido recibidas por todas las demás, entonces el estado en todas las réplicas debe ser el mismo.

3.2.1. Basadas en operaciones

Estas implementaciones comunican a las otras réplicas sólo las últimas operaciones que la réplica que envía el mensaje realizó sobre el RDT en cuestión.

Una ventaja de esto es que se reduce el tamaño de los estados que cada réplica debe mantener y el tamaño de los mensajes que estas envían. Por otro lado, estos beneficios vienen a costa de sacrificar otras propiedades como ser la velocidad de propagación de los cambios realizados, y que no son tolerantes a desorden, pérdida y/o duplicación de mensajes, siendo cada réplica la que deba encargarse de subsanar estas faltas.

Ejemplo 22 (*Contador* basado en operaciones)**.**

$$
\begin{array}{ll}
\Sigma = \mathbb{N}_0 \times \mathbb{N}_0 & \mathrm{do}(\mathrm{read}, \langle a, d\rangle, t) = (\langle a, d\rangle, a) \\
M = \mathbb{N}_0 & \mathrm{do}(\mathrm{inc}, \langle a, d\rangle, t) = (\langle a+1, d+1\rangle, \bot) \\
\vec{\sigma_0} = \lambda r.\langle 0, 0\rangle & \mathrm{send}(\langle a, d\rangle) = (\langle a, 0\rangle, d) \\
 & \mathrm{receive}(\langle a, d\rangle, d') = \langle a + d', d\rangle
\end{array}
$$

Cada réplica guarda un estado que se compone de un par $\langle a, d\rangle$ donde a es el estado actual del *contador* y d es la cantidad de incrementos realizados desde el último *send*. Inicialmente $\vec{\sigma_0}$ es $\langle 0, 0\rangle$ puesto que no han habido *incs*. Luego, *read* no modifica ninguna de las variables a y d mientras que *inc* incrementa ambas en 1.

Cada vez que se envía el estado actual de cada réplica (el par $\langle a, d\rangle$), *send* reinicia el valor de d puesto que ya se han comunicado esos últimos incrementos. Finalmente cuando se recibe un nuevo estado (*receive*), se suma dicho estado al valor del *contador* para ahora tenerlo actualizado. □

3.2.2. Basadas en estados

Las implementaciones basadas en estados comunican a todas las otras réplicas **todas** las operaciones de las que la réplica que envía el update tiene conocimiento.

Debido a esto, estas implementaciones convergen a un estado consistente mucho más rápido que las implementaciones basadas en operaciones. Esto es debido a que son *transitively delivering*, es decir, que los updates se propagan también de manera indirecta; a través de otras réplicas que re-propagan los cambios recibidos al momento de propagar su propio estado. Además, no tienen problemas con la pérdida, duplicación o reordenamiento de mensajes debido a que toda la información siempre es completa.

Como desventaja tenemos que estas implementaciones requieren más espacio de almacenamiento para almacenar los estados de cada RDT y que los mensajes que se transmiten entre las réplicas incrementan su tamaño.

Ejemplo 23 (*Contador* basado en estados)**.**

$$
\begin{array}{lcl}
\Sigma & = & \text{ReplicaID} \times (\text{ReplicaID} \to \mathbb{N}_0) \\
\vec{\sigma_0} & = & \lambda r.\langle r, \lambda s.0\rangle \\
M & = & \text{ReplicaID} \to \mathbb{N}_0 \\
\text{do(read, } \langle r, v\rangle, t) & = & (\langle r, v\rangle, \sum\{v(s) | s \in \mathit{ReplicaID}\}) \\
\text{do(inc, } \langle r, v\rangle, t) & = & (\langle r, v[r \mapsto v(r) + 1]\rangle, \bot) \\
\text{send}(\langle r, v\rangle) & = & (\langle r, v\rangle, v) \\
\text{receive}(\langle r, v\rangle, v') & = & \langle r, (\lambda s.max\{v(s), v'(s)\})\rangle
\end{array}
$$

Cada réplica guarda como estado el número de réplica que le corresponde (r) y una función v que indica, para cada réplica r_i, la cantidad de incrementos que esta hizo ($v(r_i)$). Inicialmente esta función retorna 0 para toda réplica.

Al momento de realizar un *read*, se retorna la suma de todos los incrementos hechos por todas las réplicas, es decir, la suma de todo el conjunto imagen de la función v. Una operación de *inc* hecha en la réplica r sólo modifica la función v para incrementar en 1 el valor de $v(r)$.

Finalmente, la función *send* sólo envía todo el estado completo mientras que *receive* une el estado actual con el estado recibido, preservando siempre el valor más alto entre ambos para cada réplica. Dado que la cantidad de incrementos siempre aumenta (no se puede deshacer un incremento), el comportamiento de *receive* es correcto puesto que el valor almacenado en $v(s)$ para una réplica arbitraria s siempre refleja un prefijo de la secuencia de incrementos realizados en la réplica s. □

En este trabajo nos concentramos en una implementación de un *contador* replicado basada en estados.

4. RDTS EN COQ

En este capítulo veremos cómo modelar en Coq las estructuras y operaciones que son necesarias para demostrar la correctitud de una implementación respecto de una especificación, puntualizando en una implementación basada en estados para el tipo de datos *contador*, utilizando para ello la especificación antes definida en el ejemplo 18, $\mathcal{S}_{ctr}$. De aquí en más, siempre que hablemos de especificación, estaremos refiriéndonos a la especificación del enfoque funcional.

Primero hablaremos sobre cómo modelar el LTS abstracto del *contador*, $\mathcal{T}_{\mathcal{S}_{ctr}}$, el cual se corresponde con la especificación $\mathcal{S}_{ctr}$ (ejemplo 20). Para esto mostraremos el modelado de los estados del mismo junto con las estructuras necesarias, así como las condiciones que todo LTS abstracto debe cumplir en cuanto a sus estados y sus transiciones (definición 13). A continuación revisaremos cómo modelar el LTS concreto del *contador*, $\mathcal{C}_{\mathcal{S}_{ctr}}$, correspondiente a una implementación dada del mismo (ejemplo 21), es decir, cómo modelar las condiciones que deben cumplirse para que sus transiciones sean válidas.

Finalmente daremos el modelado en Coq de la relación que debe cumplirse entre el LTS abstracto, $\mathcal{T}_{\mathcal{S}_{ctr}}$, y el LTS concreto, $\mathcal{C}_{\mathcal{S}_{ctr}}$, para que la implementación del *contador* asociada a $\mathcal{C}_{\mathcal{S}_{ctr}}$ sea correcta respecto de la especificación $\mathcal{S}_{ctr}$ asociada a $\mathcal{T}_{\mathcal{S}_{ctr}}$.

4.1. Modelado del LTS abstracto del *contador*, $\mathcal{T}_{\mathcal{S}_{ctr}}$, en Coq

Como hemos visto en 3.1.4, los estados de un tipos de datos replicado se representan en el LTS abstracto con una tupla $\langle G, P\rangle$, donde G es un $LDAG$ y P un *path*, y con un cierto conjunto de operaciones.

Dicho esto, nos proponemos primero en la subsección 4.1.1 modelar el tipo de datos para el estado abstracto de un *contador*, para lo cual necesitaremos previamente modelar los $LDAGs$ y los *paths*, e inmediatamente después, en la sección 4.1.2, mostraremos cómo modelamos el conjunto de operaciones que definen al tipo de datos.

Finalmente, para el modelado final del LTS, modelaremos las condiciones para que un estado del LTS abstracto $\langle G, P\rangle$ sea válido y para que exista una transición en el mismo, tal como vimos en la definición 13. Para la primera necesitaremos previamente, en 4.1.3, modelar la especificación presentada en el ejemplo 18 para comprobar que $P \in \mathcal{S}_{ctr}(G)$. En la subsección 4.1.4 veremos como se modelan las mencionadas condiciones.

4.1.1. Modelado del estado de un *contador*

Veremos primero el modelado de los $LDAGs$ y *paths* para luego presentar el modelado del estado en cuestión.

Para el modelado de los $LDAGs$ haremos uso nuevamente de los tipos de datos inductivos. Definiremos a un $LDAG$ como o bien vacío (no tiene ningún evento) o bien como un evento que se agrega a un $LDAG$ ya existente, indicando el label del evento y los eventos

preexistentes de los que el nuevo evento tendrá visibilidad:

```
Inductive graph : Type :=
   | Nil
   | Ext (g: graph) (e: event) (label: operation) (conj: conjunto event).
```

donde `event` es un renombre del tipo de datos `nat`, `conjunto event` es un tipo de datos representando conjuntos de eventos y `operation`, que actúa como etiqueta, es el tipo de las operaciones que definen al tipo de datos en cuestión. Veremos a continuación en 4.1.2 cuáles son las que definen al *contador*.

De esta forma `Nil` representa un $LDAG$ sin eventos mientras que, por ejemplo, `Ext (Ext Nil 1 inc empty) 2 (read 1) (add 1 empty)` representa un grafo con 2 eventos, el primero con etiqueta `inc` y el segundo con etiqueta `read,1` que tiene conocimiento del primer evento.

Definiremos también el proyector de los eventos de un grafo `g`, que nos será de utilidad de aquí en más:

```
Fixpoint events (g: graph) : conjunto event :=
   match g with
      | Nil => empty
      | Ext g' e _ _ => add e (events g')
   end.
```

Es claro que con esta definición todos los grafos que creemos serán etiquetados y dirigidos (puesto que esa es la semántica del parámetro `conj`), pero no tenemos garantías de que sean acíclicos (propiedad necesaria para ser $LDAGs$), por lo que crearemos una función que nos permita probar si el grafo construído mediante este tipo es acíclico o no:

```
Fixpoint isAnAcyclicGraph (g: graph) : Prop :=
   match g with
      | Nil => True
      | Ext g' e l c => (not (containsSet e (events g'))) ∧
                        (includesSet (events g') c) ∧
                        (isAnAcyclicGraph g')
   end.
```

La función dicta que si el grafo es vacío, entonces es trivialmente acíclico. En caso de ser la extensión de un grafo subyacente `g'` con un evento `e`, será acíclico si `e` es un nuevo evento (`e` $\notin events(g')$), las visibilidades del nuevo evento son eventos existentes en `g'` (`c` $\subseteq events(g')$) y el grafo subyacente `g'` era ya un grafo acíclico. Las funciones `containsSet` e `includesSet` se comportan como lo esperado.

Para el caso de los *paths* la situación es similar. En nuestro contexto un *path* es un $LDAG$ que exige una condición más fuerte sobre las aristas del grafo que el hecho de ser acíclico, y es que se pueda linealizar (definición 4). Esto es equivalente a que pedir que cada vez que el grafo se extienda con un nuevo evento `e`, `e` tenga visibilidad de **todos** los

eventos del grafo subyacente. Nuevamente, por definición del tipo de datos `graph`, el *path* resultante será también etiquetado y dirigido.

Definimos entonces la función `isPath`, que permite demostrar si un `graph` dado es un *path* o no:

```
Fixpoint isPath (p: graph) : Prop :=
   match p with
      | Nil => True
      | Ext p' e l c => (not (containsSet e (events p'))) ∧
                        (equalSets (events p') c) ∧
                        (isPath p')
   end.
```

Un grafo `p` será un *path* si es o bien vacío, o bien es una extensión de un grafo subyacente `p'` con un nuevo evento `e` tal que `e` $\notin$ $events(p')$, $events(p') = c$ y `p'` es también un *path*. La función `equalSets` se comporta como lo esperado, chequeando la igualdad observacional de dos conjuntos de eventos.

Finalmente, procedemos ahora a modelar la tupla $\langle G, P\rangle$ que representa un estado del *LTS* abstracto. Nuevamente haremos uso del tipo nativo *tupla* de Coq para definir nuestro nuevo tipo `abstractState`:

```
Definition abstractState : Type := graph * graph.
```

donde la semántica es que la primer componente será el grafo de visibilidad G y la segunda será el *path* de arbitración P.

Utilizaremos en la sección 4.1.4 las funciones `isAnAcyclicGraph` y `isPath` para garantizar que un `abstractState` está bien formado.

4.1.2. Operaciones del RDT *contador*

Veamos ahora cómo modelar las operaciones que definen al tipo de datos *contador*.

Recordando del capítulo 3, las posibles operaciones de un *contador* son las operaciones de $read, k$ (lectura) e inc (incremento). Al modelado de estas dos operaciones le agregaremos también las operaciones de sincronización entre réplicas $send, \langle r, v\rangle$ y $receive, \langle r', v'\rangle$, donde $\langle r, v\rangle$ y $\langle r', v'\rangle$ son instancias de estados concretos del *contador*, los cuales constaban del id de la réplica en la primera componente y de un mapeo indicando cuántos incrementos ($incs$) había hecho cada réplica en la segunda componente.

```
Inductive operation : Type :=
   | read (k: nat)
   | inc
   | send (e: concreteState)
   | receive (e: concreteState).
```

donde `concreteState` es un tipo definido como un renombre del tipo de datos tupla nativo de Coq, conteniendo a una réplica id en la primer componente y al mapeo recién mencionado en la segunda, formando así el estado del *contador* $\langle r, v\rangle$:

```
Definition concreteState : Type := replicaId * (map_replicaId_nat).
```

El tipo `replicaId` es sólo un renombre del tipo `nat` mientras que `map_replicaId_nat` es un tipo de datos representando un diccionario: `replicaId` $\mapsto$ `nat`.

4.1.3. Especificación del *contador*, $\mathcal{S}_{ctr}$

Lo que nos interesa es poder chequear si dada una arbitración P y un grafo de visibilidad G, $P \in \mathcal{S}_{ctr}(G)$.

A priori podríamos definir una función `satisfiesSctr (g p: graph)` que chequea lo que la especificación del *contador*, $\mathcal{S}_{ctr}$ pide para que $P \in \mathcal{S}_{ctr}(G)$ (ver ejemplo 18), pero hay condiciones implícitas por definición general de especificación (Definición 12) que también deberemos chequear en el modelado en Coq de la misma para que esté bien definida. Las condiciones que finalmente debemos chequear para que efectivamente $P \in \mathcal{S}_{ctr}(G)$ son las listadas a continuación:

- P debe ser un *path*: Por definición de especificación, el codominio de $\mathcal{S}_{ctr}$ es $2^{\mathbb{P}(L)}$
- G debe ser un $LDAG$: Por definición de especificación, el dominio de $\mathcal{S}_{ctr}$ es $\mathbb{G}(L)$
- Los eventos de G y P deben ser los mismos y estar etiquetados idénticamente: Por definición de especificación, $\mathcal{S}_{ctr}(G) \in 2^{\mathbb{P}(E_G, \lambda_G)}$
- P debe cumplir las condiciones de la especificación del *contador* para pertenecer a $\mathcal{S}_{ctr}(G)$

Dicho esto, procedemos a modelar estos requerimientos con la siguiente *Prop*:

```
Definition isAdmissibleInSctr (g p: graph) : Prop :=
      isPath p ∧
      isAnAcyclicGraph g ∧
      sameEventsAndLabels g p ∧
      satisfiesSctr g p.
```

donde cada *Prop* de la conjunción se corresponde con las condiciones recién mencionadas. Las *Props* `sameEventsAndLabels` y `satisfiesSctr` comprueban respectivamente que los eventos y etiquetas en `g` y en `p` sean los mismos y que se cumpla lo que la definición de la especificación $\mathcal{S}_{ctr}$ pide. Vemos a continuación el modelado en Coq de las mismas.

```
Definition sameEventsAndLabels (g other: graph) : Prop :=
      (equalSets (events g) (events other)) ∧
      forall e: event, (containsSet e (events g))
               -> ((label e other) = (label e g))).
```

Se pide que los conjuntos de eventos de ambos grafos sean observacionalmente iguales y que para cada evento `e` $\in events(g)$, la etiqueta de `e` en `g` se corresponda con la etiqueta de `e` en el otro grafo `other`.

Por último llegamos a la *Prop* que efectivamente chequea que la condición enunciada en el ejemplo 18 por $\mathcal{S}_{ctr}$ se cumpla:

```
Fixpoint satisfiesSctr (p g: graph) : Prop :=
   match g with
      | Nil => True
      | Ext g' e (read k) c => (satisfiesSctr p g') ∧ (k = incs g' c)
      | Ext g' e _ c => satisfiesSctr p g'
   end.
```

que, como se puede apreciar, no hace uso del parámetro `p`, tal como ocurre en la definición de $\mathcal{S}_{ctr}$. Sólo se chequean condiciones sobre `g`, para ver si todos los $read, k$ tienen visibilidad de exactamente k *incs*. Para esto se revisan los eventos en orden inverso a como fue extendido el grafo, si el grafo es vacío entonces es trivialmente cierto que satisface la condición ya que no hay ningún $read, k$. Si se encuentra con un evento que no es un $read, k$, procedemos a saltearlo y a seguir con los eventos interiores, ya que sólo nos interesan los eventos de este tipo. Al encontrarnos con un evento etiquetado con $read, k$, definiremos que se satisface la *Prop* sobre $\mathcal{S}_{ctr}$ si k es igual a la cantidad de *incs* que existen sobre los eventos en `c` en el grafo `g'` (ya que `c` son los eventos de los que el $read, k$ tiene visibilidad) y si el subgrafo `g'` también satisface la condición de $\mathcal{S}_{ctr}$ recursivamente.

La función `inc` solamente cuenta la cantidad de *incs* del conjunto de eventos `c` que recibe como parámetro. A modo de completitud se detalla la función a continuación:

```
Fixpoint incs (g: graph) (events: conjunto event) : nat :=
   countIncs (map (fun e' =>label e' g) (toList events)).

Fixpoint countIncs (l: list operacion) : Prop :=
   match l with
      | nil => 0
      | inc :: l' => 1 + (countIncs l')
      | _ :: l' => countIncs l
   end.
```

4.1.4. *LTS* abstracto del *contador*

Recordando la definición 13 de *LTS* abstracto, e instanciándola con $\mathcal{S}_{ctr}$, tenemos dos condiciones que deben cumplirse en el mismo, las cuales son:

- $\langle G, P\rangle$ es un estado si y solo si $P \in \mathcal{S}_{ctr}(G)$
- $\langle G, P\rangle \xrightarrow{\ell} \langle G', P'\rangle$ es una transición siempre que $G' = G^{\ell}$ y $P'|_{E_G} = P$ para toda etiqueta ℓ, que en nuestro caso serán las operaciones del *contador*.

Ahora que tenemos un tipo de datos para los estados del *LTS* abstracto del *contador*, `abstractState`, necesitamos modelar la *Prop* `consistentState`, que chequee que un estado dado es un estado válido de $\mathcal{T}_{\mathcal{S}_{ctr}}$, es decir, que cumple con la primera regla. Puesto que ya tenemos modelado cómo chequear si $P \in \mathcal{S}_{ctr}(G)$, es sólo cuestión de aplicarla:

```
Definition consistentState (s: abstractState) : Prop :=
   isAdmissibleInSctr (fst s) (snd s).
```

donde `fst` y `snd` son los proyectores de cada coordenada de `s`.

Para la segunda regla, utilizaremos las antes explicadas *Inductive propositions*. Al igual que en el ejemplo 12, modelaremos cuándo una transición es valida en el LTS abstracto $\mathcal{T}_{\mathcal{S}_{ctr}}$ según la segunda regla de la definición 13.

Para esto necesitaremos programar las operaciones de extensión y de restricción (definición 5) para ver respectivamente que $G' = G^{\ell}$ y que $P'|_{E_G} = P$.

La primera es trivial dado que la forma de construcción de los grafos es por extensión de grafos previamente construídos. La modelaremos como una función solo para obviar la indicación de que el conjunto `c` de eventos que el nuevo evento conoce son todos los eventos del grafo.

```
Definition extend (e: event) (l: operation) (g: graph) : graph :=
   Ext g e l (events g).
```

La segunda la modelaremos como una función que proyecta todos los eventos de un grafo a excepción de cierto evento k, que vendrá a ser el nuevo evento agregado al grafo G que da origen al grafo G'. Esto es análogo a calcular la restricción $|_{E_G}$ puesto que los eventos de P' son los eventos de G' (pues $\langle G', P' \rangle$ es también un estado válido del LTS), por lo que los eventos de $P'|_{E_G}$ serán todos los eventos de G' a excepción de k. Dado que dicha función de proyección mantendrá el orden relativo de los nodos, el resultado de la misma será P. La función es la siguiente:

```
Fixpoint projectWithout (g: graph) (e: event) : graph :=
   match g with
      | Nil => Nil
      | Ext g' e' l' c' =>
         if eqb e e'
         then g'
         else Ext (projectWithout g' e) e' l' (remove e c')
   end.
```

donde `remove` es una función que remueve el elemento `e` del conjunto `c'`.

Presentadas las dos funciones necesarias, modelamos ahora entonces, cuándo una transición es válida en el LTS abstracto:

```
Inductive counterAbstractLTS:abstractState->operation->abstractState->Prop:=
      | counterAbstractLTS_transition (s: abstractState) (o: operation)
                                      (s': abstractState) (k: event):
            (consistentState s) ∧
            (consistentState s') ∧
            (equalsGraphs (extend k o (fst s)) (fst s')) ∧
            (equalsGraphs (snd s) (projectWithout (snd s') k)) ->
```

```
counterAbstractLTS s o s'.
```

Interpretando esta *Inductive Proposition*, una transición $s \xrightarrow{o} s'$ en el LTS abstracto es válida para dos estados abstractos s y s' y para una operation o si:

- s es un estado consistente, es decir, si $s = \langle G, P \rangle$, entonces $P \in \mathcal{S}_{ctr}(G)$
- s' es también un estado consistente, es decir, si $s' = \langle G', P' \rangle$, entonces $P' \in \mathcal{S}_{ctr}(G')$
- Si $s = \langle G, P \rangle$ y $s' = \langle G', P' \rangle$, entonces G^o es observacionalmente igual a G'
- Si $s = \langle G, P \rangle$ y $s' = \langle G', P' \rangle$, entonces P es observacionalmente igual a $P'|_{E_G}$

Por simplicidad de notación, definimos una `Notation` de Coq para representar una transición válida del LTS abtsracto `counterAbstractLTS s o s'`:

```
Notation ''as1 '--' l '-- >>' as2'' := (counterAbstractLTS (as1) (l) (as2))
```

4.2. Modelado del LTS concreto del *contador*, $\mathcal{C}_{\mathcal{S}_{ctr}}$, en Coq

Como fue visto en el ejemplo 21 de la implementación basada en estados del *contador*, los estados del mismo son el conjunto $\Sigma = \mathcal{R} \times (\mathcal{R} \to \mathbb{N})$, donde $\mathcal{R}$ representa el conjunto de *replica ids*, es decir, el conjunto de identificadores de cada réplica. Nuestro LTS concreto, $\mathcal{C}_{\mathcal{S}_{ctr}}$, tendrá los mismos estados y sus transiciones representarán las diversas modificaciones que un estado dado percibirá frente a cada operación. Las reglas de transición que dicha implementación del *contador* debe cumplir son las 4 reglas detalladas en la figura del mismo ejemplo 21.

Ya hemos definido anteriormente en la sección 4.1.2 los `concreteState`s como una tupla que tiene el id de la réplica como primera componente y un mapeo de replica id en cantidad de incrementos de esa réplica como segunda componente.

Ahora solo resta modelar las transiciones válidas del LTS concreto. Para esto usaremos nuevamente una *Inductive Proposition*, definiendo un caso para cada posible transición:

```
Inductive counterLTS:concreteState -> operation -> concreteState -> Prop:=
      | counterLTS_read (e: concreteState) (k: nat):
            sumValues (snd e) = k ->
                  counterLTS e (read k) e
      | counterLTS_send (e: concreteState):
                  counterLTS e (send e) e
      | counterLTS_inc (e e': concreteState):
            fst e = fst e' ∧
            equalMaps (increment (fst e) (snd e)) (snd e') ->
                  counterLTS e inc e'
      | counterLTS_receive (e e' e'': concreteState):
            fst e = fst e'' ∧
            equalMaps (max (snd e) (snd e')) (snd e'') ->
```

```
counterLTS e (receive e') e''.
```

Para interpretar correctamente estas cuatro reglas, veámoslas de a una. La regla `counterLTS_read` indica que una transición desde un estado `e` al mismo estado `e` por una operación `read k` (`counterLTS e (read k) e`) es válida si la sumatoria de los incrementos realizados por todas las réplicas es igual a `k`. Esta condición la logramos con una función `sumValues`, que dado un mapeo, o diccionario, nos devuelve la sumatoria de todos los valores del mismo, independientemente de la key (replica id) a la que esté asociado. Dicho mapeo es la segunda componente del `concreteState e`. La regla `counterLTS_send` es la mas sencilla de las cuatro, solamente se acepta la transición si el `concreteState` de origen es el mismo que el `concreteState` de destino y el mismo que el que la operación de sincronización *send* conlleva (counterLTS e (send e) e). Para la regla de incremento `counterLTS_inc` los estados deberán ser distintos (pasando del estado `e` al estado `e'`), pero preservando dos condiciones: primero que la primer componente del estado, es decir, el id de la réplica, debe permanecer igual (por ello la condición `fst e = fst e'`) y segundo que el mapeo entre replica ids e incrementos asociado a cada estado debe ser observacionalmente igual a excepción de la cantidad de incrementos asociado a la réplica que está realizando el *inc*, el cual debe incrementar su valor en 1, puesto que esta réplica ahora realizó un *inc* más. Para comprobar esto utilizamos una función `increment`, que dado un mapeo con valores en los naturales y una key (replica id en este caso), incrementa su valor en 1. Finalmente la condición se chequea pidiendo que el mapeo asocido a `e` incrementando su valor para la clave de la réplica actual sea obsevacionalmente igual al mapeo asociado a `e'`. La función `equalMaps` es la encargada de chequear que dos mapas sean observacionalmente iguales (que a la misma clave le corresponda el mismo valor). Finalmente para la regla de receive (`counterLTS_receive`) vemos nuevamente que los estados cambiarán pero que dos condiciones deben cumplirse: la primera es igual al caso del `inc`, el id de la réplica debe mantenerse, y la segunda es que el mapeo asociado al nuevo estado `e''` debe ser observacionalmente igual al máximo entre los mapeos de `e`, el estado de origen, y `e'`, el estado recibido, es decir, la cantidad de *incs* asociada a una replica id r en el mapeo del estado `e''` debe ser igual al máximo entre el valor asociado a r para los mapeos de `e` y `e'`. Para lograr esto utilizamos una función `max`, que recibe dos mapas o diccionarios y retorna otro diccionario que tiene los máximos para cada replica id.

Nuevamente, por simplicidad de notación, definimos otra `Notation` de Coq para representar una transición válida del *LTS* concreto `counterLTS e o e'`:

```
Notation ''cs1 '==' l '==>>' cs2'' := (counterLTS (cs1) (l) (cs2))
```

4.3. Correctitud de una implementación del *RDT contador* como simulación entre $\mathcal{C}_{\mathcal{S}_{ctr}}$ y $\mathcal{T}_{\mathcal{S}_{ctr}}$

A la hora de demostrar que una implementación del *contador* es correcta respecto de la especificación del mismo, lo hacemos a través de una relación entre los estados de $\mathcal{C}_{\mathcal{S}_{ctr}}$ y $\mathcal{T}_{\mathcal{S}_{ctr}}$. Llamamos a esta relación una *implementation relation* y la notamos con $\mathcal{I}_{\mathcal{S}_{ctr}}$. Si la relación $\mathcal{I}_{\mathcal{S}_{ctr}}$ cumple ciertas propiedades, entonces decimos que la implementación es correcta respecto de la especificación.

Antes de proceder a detallar cuáles son y cómo modelar dichas propiedades, veamos primero cómo modelar relaciones en Coq para poder modelar el tipo de las relaciones entre los estados mencionados.

4.3.1. Relación $\mathcal{I}_{\mathcal{S}_{ctr}}$ en Coq

Hay más de una forma de representar las mismas, nosotros optamos por utilizar aquella que dados 2 elementos de la relación, nos devuelve una *Prop* que será demostrable si los elementos están relacionados. Para definir nuestras relaciones entre estados abstractos y concretos definiremos primero las relaciones genéricas que usaremos:

```
Definition relation (X Y: Type) := X -> Y -> Prop.
```

Esta definición nos permite instanciar los distintos tipos de relaciones que necesitemos, como por ejemplo, el tipo de las relaciones entre estados del LTS concreto del *contador*, $\mathcal{C}_{\mathcal{S}_{ctr}}$, y estados del LTS abstracto del mismo, $\mathcal{T}_{\mathcal{S}_{ctr}}$:

```
Definition relIS : Type := relation concreteState abstractState.
```

la cual usaremos luego para instanciar la relación que será necesaria para demostrar que una implementación del *contador* que sigue a $\mathcal{C}_{\mathcal{S}_{ctr}}$ es correcta respecto de su especificación.

4.3.2. Propiedades sobre $\mathcal{I}_{\mathcal{S}_{ctr}}$ para garantizar la correctitud de una implementación

Ahora que hemos definido las relaciones, estamos listos para hablar de las condiciones que una *Implementation relation* $\mathcal{I}_{\mathcal{S}_{ctr}}$ debe cumplir para que la implementación de un *RDT contador* sea correcta. Podrían existir diversas relaciones $\mathcal{I}_{\mathcal{S}_{ctr}}$, pero la existencia de una que cumpla con todas las propiedades descritas a continuación garantiza la correctitud de la implementación.

Utilizaremos σ, σ' y σ'' para referirnos a estados del LTS concreto $\mathcal{C}_{\mathcal{S}_{ctr}}$ y $\langle \bullet, \bullet \rangle$ para referirnos a estados del LTS abstracto $\mathcal{T}_{\mathcal{S}_{ctr}}$.

La relación $\mathcal{I}_{\mathcal{S}_{ctr}}$ es una relación entre los estados de $\mathcal{C}_{\mathcal{S}_{ctr}}$ y $\mathcal{T}_{\mathcal{S}_{ctr}}$ tal que si $(\sigma, \langle G, P \rangle) \in \mathcal{I}_{\mathcal{S}_{ctr}}$, entonces para todo σ', σ'', op y v se cumple que:

1. Si $\sigma \xrightarrow{op,v} \sigma'$ entonces $\exists G', P'$ tales que $\langle G, P \rangle \xrightarrow{op,v} \langle G', P' \rangle$ y $(\sigma', \langle G', P' \rangle) \in \mathcal{I}_{\mathcal{S}_{ctr}}$

2. Si $\sigma \xrightarrow{rcv,\sigma'} \sigma''$ y $(\sigma', \langle G', P' \rangle) \in \mathcal{I}_{\mathcal{S}_{ctr}}$ entonces

   $\exists P''$ tal que $P'' \in P \bigotimes P'$ y $(\sigma'', \langle G \sqcup G', P'' \rangle) \in \mathcal{I}_{\mathcal{S}_{ctr}}$

3. Si $\sigma \xrightarrow{send,\sigma} \sigma$ entonces $\exists G', P'$ tales que $(\sigma, \langle G', P' \rangle) \in \mathcal{I}_{\mathcal{S}_{ctr}}$ y $G \sqcup G' = G$ y

   $P \bigotimes P' = \{P\}$

donde op, v son las operaciones que no son de sincronización, junto con su valor de respuesta. Para el caso del *contador*, las operaciones op son solamente *read* e *inc*, junto con los valores de respuesta v para cada uno, que son los números naturales con el cero y *ok* respectivamente.

Nótese que esta versión es levemente distinta a la forma más general de correctitud de implementaciones respecto de especificaciones presentada en [1], puesto que aquí nos estamos restringiendo a la correctitud del tipo de datos replicado *contador*, que tiene sus particularidades que nos permiten alterar dicha forma más general. Por ejemplo, que el mismo carece de transiciones internas τ, por lo que nos permite no agregar una regla para este caso, o también, que el parámetro asociado al send es σ ya que así es como funciona dicha transición de la implementación del *contador*, mientras que en otros *RDTs* este valor podría ser otro estado concreto σ'.

A continuación veremos cómo implementar en Coq cada una de estas reglas, pero primero veremos la regla general que, dada una `relIS`, nos permite demostrar si las 3 reglas se cumplen o no, demostrando así si la implementación del *contador* es correcta o no:

```
Definition isCorrectImplementation (IS: relIS) : Prop :=
   forall (p: concreteState) (q: abstractState),
      ((consistentState q) ∧ (IS p q)) -> (
         (op_relation_step IS p q) ∧
         (receive_relation_step IS p q) ∧
         (send_relation_step IS p q)
      ).
```

donde `op_relation_step`, `receive_relation_step` y `send_relation_step` son las funciones que comprueban cada una de las tres reglas correspondientemente (serán detalladas luego). La función refleja la teoría, si dos estados `p` y `q` están relacionados vía una relación `IS` (`IS p q`) del tipo `relIS`, entonces deben cumplirse las 3 propiedades. Cada una de estas propiedades hará uso de las variables universales (como σ') que necesite. Nuevamente, debemos salvaguardar en el modelado un aspecto que en la teoría está implícito, y es que el estado abstracto de origen, $\langle G, P\rangle$, debe ser un estado válido. Esto lo agregamos a las hipótesis pidiendo que `q` sea un estado consistente (`consistentState q`).

Ahora veamos el detalle de las tres funciones asociadas a cada una de las tres reglas.

Empecemos por la regla n°1, la regla de operación `op_relation_step`. Lo que esta regla exige es que, si $(\sigma, \langle G, P\rangle)$ estaban relacionados en $\mathcal{I}_{\mathcal{S}_{ctr}}$ y ocurre una transición en $\mathcal{C}_{\mathcal{S}_{ctr}}$ a raíz de cierta transición op, v que lo lleva desde el estado σ al estado σ', deberá entonces existir también una transición en el $\mathcal{T}_{\mathcal{S}_{ctr}}$ con la misma etiqueta, y los estados correspondientes a los que ambos *LTSs* transicionen deberán estar relacionados también. Para el caso del *contador*, esto se refleja en que G' y P' tendrán ahora un evento asociado a la etiqueta op, v, y la misma debe cumplir que si es un $read, k$, la cantidad de *incs* que este ve en G' es exactamente k. De lo contrario, $P' \notin \mathcal{S}_{ctr}(G')$ y por lo tanto $\langle G', P'\rangle$ no sería un estado válido del $\mathcal{T}_{\mathcal{S}_{ctr}}$, imposibilitando así la transición pedida en $\mathcal{T}_{\mathcal{S}_{ctr}}$ para la regla n°1.

Veamos su modelado en Coq:

```
Definition op_relation_step (IS: relIS) (σ: concreteState) (q: abstractState):Prop:=
  forall (o: operation) (σ': concreteState),
    ((not (isASyncOperation o)) ∧ (σ == o ==>> σ')) ->
      (exists (q': abstractState), (q -- o -- >> q') ∧ (IS σ' q')).
```

Recordar que, en nuestro modelado, una `operation` ya contiene tanto el nombre de la operación como su valor. Se puede apreciar que como hipótesis se asume que existe una transición en el LTS concreto entre σ y σ' (hemos utilizado la `Notation` antes definida para esto). Como consecuente, se exige que exista un estado abstracto `q'`, tal que el LTS abstracto también transicione hacia `q'` y que σ' y `q'` estén relacionados por la relación `IS` de tipo `relIS`. Como ya hemos mencionado, esta regla debe cumplirse sólo para las operaciones que no son de sincronización (para el caso del *contador*, *read* e *inc*), por lo que también se asume entre las hipótesis que `o` no es una operación de sincronización.

Prosigamos con la regla de `receive_relation_step`. La misma especifica que si estamos en un estado σ y, al recibir un estado σ' nos movemos hacia un nuevo estado σ'', cuando ademas ese estado recibido σ' está bien relacionado con cierto estado abstracto $\langle G', P' \rangle$ en $\mathcal{I}_{\mathcal{S}_{ctr}}$, entonces lo que debe ocurrir es que exista un P'' tal que sea conformado por una intercalación de los elementos de P y P' preservando el orden relativo de los eventos de estos. Además, el nuevo estado σ'' debe estar relacionado en $\mathcal{I}_{\mathcal{S}_{ctr}}$ con $\langle G \sqcup G', P'' \rangle$ para garantizar que la unión de los grafos de visibilidad y de los *paths* que teníamos, con los recibidos, no nos llevan a elementos fuera de la relación $\mathcal{I}_{\mathcal{S}_{ctr}}$.

Nuevamente, debemos considerar algunos aspectos en el modelado. Por precondición en la definición del operador $\bigotimes$, los mismos eventos en ambos *paths* deben tener las mismas etiquetas. Dicha condición se asume dada por definición del operador en la teoría, pero nosotros debemos aclararla explícitamente en el modelado. Así mismo, para garantizar que exista un $P'' \in P \bigotimes P'$, es necesario que P y P' no sean contradictorios, es decir, que el orden relativo de los eventos que están en ambos *paths* sea el mismo en ambos, lo agregaremos pidiendo que los subgrafos comunes sean iguales, lo cual, en el caso de los *paths*, equivale a pedir que la proyección sobre los eventos en la intersección genere dos *paths* observacionalmente iguales. De manera similar, la definición del operador $\sqcup$ asume que los dos grafos G y G' son *compatibles*, lo cual quiere decir que la historia en común vista por ambos no es inconsistente. Debemos asumirlo también para que la aplicación del operador esté bien definida. Por último, G', el grafo de visibilidad asociado al estado σ' recibido, debe ser también un $LDAG$, P' debe ser también un *path* y, en particular, uno válido para G'. En definitiva, que sean un `consistentState`, por lo cual también lo agregamos a las hipótesis de la función.

A continuación, el modelado en Coq:

```
Definition receive_relation_step (IS: relIS) (σ: concreteState) (q: abstractState):Prop:=
  forall (σ σ': concreteState) (q': abstractState), (
    let G := fst q in
    let P := snd q in
    let G' := fst q' in
    let P := snd q' in
```

```
        ((σ == (receive σ') ==>> σ'') ∧
        (repeatedEventsHaveSameLabels P P') ∧
        (compatibleLDAGs G G') ∧
        (consistentState q') ∧
        (commonSubgraphsAreEqual P P') ∧
        (IS σ' q')) ->
            (exists (P'': graph),
                (P'' ∈ P ⨂ P') ∧
                (IS σ'' ((G ⊔ G'), P''))
            )
    )
```

donde la sentencia `let ... in` nos permite agregar claridad a la función, renombrando las componentes de los estados abstractos `q` y `q'` para un mejor entendimiento y mapeo con la teoría. Las funciones `repeatedEventsHaveSameLabels`, `compatibleLDAGs` y `commonSubgraphsAreEqual` son funciones que retornan *Props* y que realizan los chequeos anteriormente comentados. En el consecuente, la notación $\bullet\in\bullet\bigotimes\bullet$ es solo azúcar sintáctico utilizando `Notations` de Coq para claridad, pero por debajo esto se resuelve a una función `inCrossProduct` que comprueba si un *path* dado pertenece al producto $\bigotimes$ entre dos *paths* (esto sólo podrá ser cierto si P'' es un *path*, por lo que no hay necesidad de comprobar esa condición por separado).

Por último, veamos la tercer regla, `send_relation_step`. Esta regla pide que si se envía un estado σ, este sea un estado concreto válido en la relación $\mathcal{I}_{\mathcal{S}_{ctr}}$, es decir, que exista un estado abstracto con el que se relacione. Además se pide que el $LDAG$ del estado abstracto con el que se relacione no tenga más información que el $LDAG$ del estado abstracto con el que el estado concreto actual (también σ) se relaciona. Esto se garantiza pidiendo que la unión compatible de ambos $LDAGs$ dé como resultado el $LDAG$ actual (si G' tuviera eventos que G no tiene, la unión compatible no sería G). Lo mismo ocurre con P y P', para pedir que no haya eventos en P' que no estén en P se exige que el producto de ambos dé como único posible *path* a P.

Veamos la implementación en Coq:

```
Definition send_relation_step (IS: relIS) (σ: concreteState) (q: abstractState):Prop:=
    (σ == (send σ) ==>> σ) ->
        let G := fst q in
        let P := snd q in
            (exists (G' P': graph),
                (
                    (IS σ (G', P')) ∧
                    (equalsGraphs (G ⊔ G') G) ∧
                    (pathIsASubOrden P' P)
                )
            ).
```

donde la condición $P \bigotimes P' = \{P\}$ se chequea llamando a `pathIsASubOrden` P' P, ya que el hecho de que $P \bigotimes P' = \{P\}$ es equivalente a que P' sea un suborden de P.

4.3.3. Relación $\mathcal{I}_{\mathcal{S}_{ctr}}$ elegida y teorema para la correctitud

Como mencionamos previamente, para que la implementación del *contador* sea correcta respecto de su especificación $\mathcal{S}_{ctr}$ debe existir una relación `relIS` que cumpla las reglas recién presentadas. Reutilizaremos la relación $\mathcal{I}_{\mathcal{S}_{ctr}}$ propuesta en [1] como instancia de una relación de `relIS` y, luego, daremos el teorema que dicta que dicha relación demuestra la correctitud de la implementación del *contador* en base a su especificación como simulación de $\mathcal{C}_{\mathcal{S}_{ctr}}$ a $\mathcal{T}_{\mathcal{S}_{ctr}}$.

Por simplicidad recapitulamos la relación presentada en [1]:

$$\mathcal{I}_{\mathcal{S}_{ctr}} = \{(\langle r, v\rangle, \langle G, P\rangle) \mid \text{existe } f : E_G \mapsto \mathcal{R} \text{ tal que } \forall r \in \mathcal{R}, v(r) = \#\{e \mid f(e) = r \text{ y } \lambda(e) = \langle inc, ok\rangle\}\ \}$$

Lo que la relación plantea es que dos estados concreto y abstracto estarán relacionados siempre y cuando podamos asignar cada evento del grafo de visibilidad G a una réplica (como forma de identificar qué réplica dio origen a cada evento) y luego pidiendo que el mapeo entre réplica y cantidad de incrementos (v) del estado concreto sea consistente con la cantidad de incrementos que efectivamente realizó cada réplica. Es decir, si la réplica número 1 dio origen a 5 distintos eventos de incremento ($\langle inc, ok\rangle$), entonces $v(1)$ deberá ser 5. Esto lógicamente se pide que valga para todas las réplicas. De esta forma aseguramos que el estado concreto es consistente en la información que contiene con respecto a todos los eventos que existen en G, el grafo de visibilidad asociado a un estado abstracto.

Luego procedimos a modelar dicha relación $\mathcal{I}_{\mathcal{S}_{ctr}}$ en Coq:

```
Definition counterRelation : relIS := fun concr abstr =>
   exists (f: (event -> replicaId)),
      forall (r: replicaId),
         find (snd concr) r = incsOfReplica f r (fst abstr).
```

donde `find` es una función que, dado un `map_replicaId_nat` y una `replicaId`, devuelve el valor asociado en el mapeo. Por otro lado, `incsOfReplica` es la función que computa cuál es la totalidad de los incrementos realizados para una réplica dada `r`. La misma es la que sigue:

```
Fixpoint incsOfReplica (f: event -> replicaId)(r: replicaId)(g: graph):nat:=
   match g with
   | Nil => 0
   | Ext g' e inc c => (if (eqb (f e) r) then 1 else 0)+incsOfReplica f r g'
   | Ext g' e _ c => incsOfReplica f r g'
   end.
```

La función simplemente itera por todos los eventos del grafo `g` en busca de operaciones de incremento. En caso de encontrar una, se preguntará si la misma fue realizada por la réplica `r`. En caso afirmativo, sumará 1 al resultado final y proseguirá iterando el resto de los eventos. En caso negativo, simplemente proseguirá con el resto de los eventos.

Finalmente, formularemos el siguiente teorema que establece que la implementación basada en estados presentada para el *contador* en la sección 4.3 es correcta respecto de la especificación $\mathcal{S}_{ctr}$. Para esto utilizaremos la ya presentada *Prop* `isCorrectImplementation`:

```
Theorem counterImplementationIsCorrect : isCorrectImplementation counterRelation.
```

La demostración del mismo sigue los lineamientos desarrollados en [1] pero vía la utilización de las tácticas de Coq. Si bien explicaremos la idea general, los inconvenientes encontrados y las etapas de la demostración, no pretendemos dar aquí un detalle exhaustivo de la misma, lo cual ofuscaría la lectura, ya que el objetivo de este trabajo es dar un marco de trabajo para el modelado de las estructuras y cimientos sobre los que las demostraciones de correctitud de los *RDTs* se apoyan (por ejemplo, el modelado de los *LTSs*), utilizando como ejemplo de esto al *RDT contador*. En caso de estar interesados en la demostración completa puede accederse a la misma en el repositorio[1].

Procederemos a describir las ideas generales de la demostración, la cual fue realizada por casos tal como en [1].

Primero se realizó la demostración de la regla `op_relation_step`, la cual a su vez debió dividirse en 2 casos, para el $read, k$ y para el *inc*, las cuales son las operaciones de no sincronización del *contador*. Para que la regla valga por completo, debe mostrarse que vale para ambas operaciones. Para ambos casos debimos mostrar que existen los G' y P' pedidos. Siguiendo de la demostración de [1], sabemos que estos son ni más ni menos que la extensión de G y P con un nuevo evento T etiquetado con la operación correspondiente, ya que así lo exige el *LTS* abstracto para que la transición sea válida. Por lo que le indicamos a Coq que los G' y P' pedidos eran, por ejemplo, la extensión de G y de P con un nuevo evento que conoce a todos los eventos actuales de cada uno. De esta forma también preservamos que P' siga siendo un *path* si P lo era. Luego de indicarle a Coq que estos eran los existenciales necesarios, debimos demostrar que $\langle G, P\rangle$ y $\langle G', P'\rangle$ eran estados consistentes, que G' era observacionalmente igual a extender a G con un evento fresco y que P' es una saturación con T de P. Estas 4 condiciones fueron debidas a que esa es la definición de transición válida en el *LTS* abstracto. Un inconveniente encontrado fue que no podemos indicarle a Coq que extienda los *LDAGs* con un evento no existente previamente (algo que en la teoría se pide y ya, ya que siempre existe uno nuevo), sino que tenemos que indicarle nosotros cuál es. Para subsanar esto decidimos que el nuevo evento T sería siempre igual a `máximo evento + 1`, asegurando así que T es siempre un evento nuevo.

Luego de esto debimos mostrar que σ' y $\langle G', P'\rangle$ estaban relacionados en $\mathcal{I}_{\mathcal{S}_{ctr}}$. Para esto, tal como en [1], propusimos la función f' necesaria que la $\mathcal{I}_{\mathcal{S}_{ctr}}$ requiere para que dos eventos estén relacionados y procedimos a mostrar que se cumple. La función $f' : E_{G'} \mapsto \mathcal{R}$ propuesta fue:

$$f'(e) = \begin{cases} f(e) & \text{si e} \in E_G \\ r_j & \text{si e} = \text{T} \end{cases}$$

donde r_j es la réplica que realizó la operación (`fst` σ) y f es la función que existe ya que σ y $\langle G, P\rangle$ estaban relacionados por hipótesis en $\mathcal{I}_{\mathcal{S}_{ctr}}$. La misma función en Coq se modeló

[1] https://github.com/pablogomez93/tesis-rdt/blob/master/src/CounterImplCorrect.v

como:

```
f' := fun e: event => if eqb e T then fst σ else f e
```

y mostramos que el comportamiento exigido por $\mathcal{I}_{\mathcal{S}_{ctr}}$ se cumplía. Es decir, que para todo $r \in \mathcal{R}$ coincidían los valores de la función v con la cantidad de *incs* realizados por cada réplica en G'. Para esto separamos los eventos en dos casos, cuando eran iguales a T y cuando no. Para el caso en que los eventos no son iguales a T, mostramos que `f'` y `f` se comportan de la misma manera (para esto utilizamos y demostramos un teorema `functionsBehaveTheSame`) y para el caso en que $e = T$, mostramos que la condición se cumplía específicamente para cada uno de los dos tipos de operaciones, los *read* k y los *inc*.

El siguiente paso de la demostración es la regla del receive, `receive_relation_step`. El primer paso es agregar todas nuestras hipótesis al contexto para luego demostrar que existe el *path* P'' que cumple las dos condiciones pedidas. Para demostrar que existe un $P'' \in P \bigotimes P'$ dimos de alta un teorema (`crossProductNotEmpty`), que luego usamos en esta demostración, que establece que dicho P'' existirá si P y P' son *paths*, si el orden relativo entre los eventos comunes de ambos no es contradictorio (si $e_1 < e_2$ en P, entonces $e_1 < e_2$ en P' y viceversa) y si los eventos que están en ambos *paths* tienen las mismas etiquetas en ambos. Luego de aplicar este teorema en nuestro *goal* en Coq, debimos demostrar entonces que estás cuatro condiciones se cumplían, lo cual realizamos utilizando las hipótesis presentes en nuestro contexto y usando el hecho de que $\langle G, P\rangle$ y $\langle G', P'\rangle$ eran estados consistentes por hipótesis (por lo cual se implica que P y P' son *paths*).

Luego debíamos demostrar que σ'' y $\langle G \sqcup G', P''\rangle$ estaban relacionados en $\mathcal{I}_{\mathcal{S}_{ctr}}$. Debimos definir entonces nuevamente una `f''` que cumpla las condiciones de $\mathcal{I}_{\mathcal{S}_{ctr}}$ para ambos estados concreto y abstracto. Utilizamos nuevamente la $f'' : E_{G \sqcup G'} \mapsto \mathcal{R}$ propuesta en [1], la cual es:

$$f''(e) = \begin{cases} f(e) & \text{si e} \in E_G \\ f'(e) & \text{sino} \end{cases}$$

donde f y f' son las correspondientes funciones que existen ya que $(\sigma, \langle G, P\rangle)$ y $(\sigma', \langle G', P'\rangle)$ pertenecen a $\mathcal{I}_{\mathcal{S}_{ctr}}$. El modelado en Coq de la misma es:

```
f'' := fun e: event => if (setContains e (events G)) then f e else f' e
```

donde `setContains` es una función `event -> conjunto event -> bool` que actúa como lo esperado. Para concluir que dicha función cumplía la pertenencia a la relación $\mathcal{I}_{\mathcal{S}_{ctr}}$ fuimos transformando nuestro *goal* vía tácticas de Coq y utilizando diversos lemas que nos permitieron modularizar la demostración, como `incrementJustAffectIncrementedValue`, que es un teorema que demuestra que, en un mapeo v que marca la cantidad de incrementos realizados por cada réplica, incrementar la cantidad de *incs* asociados a una réplica no afecta la cantidad de *incs* asociados a otras réplicas.

Finalmente, la regla del send, `send_relation_step`. Esta regla es la más directa, puesto que si se está en un estado concreto σ, el estado que la operación de send envía es también σ para el caso del *RDT contador*. Dicho esto, dado que se pide que exista un estado abstracto $\langle G', P'\rangle$ relacionado con σ en $\mathcal{I}_{\mathcal{S}_{ctr}}$, podemos proponer el mismo estado abstracto $\langle G, P\rangle$, que está relacionado con σ por hipótesis.

Resta entonces demostrar que $G \sqcup G'$ es observacionalmente igual a G, asumiendo ahora que G es igual a G' y que P es igual a P'. Para el primero lo hemos resuelto demostrando un teorema, `unionIdempotentOnSameGraph`, que demuestra que para todo grafo G, $G \sqcup G = G$. La demostración del mismo consistió en hacer inducción estructural en G. Para la segunda parte, demostrar que un *path* P' es suborden de otro *path* P implica demostrar que los órdenes relativos son consistentes (si $e_1 < e_2$ en P, entonces $e_1 < e_2$ en P' y viceversa) y que los eventos en P' están incluídos en los eventos de P. Dado que $P = P'$, demostrar que los órdenes son consistentes fue cuestión de demostrar que los grafos eran observacionalmente iguales, para lo cual dimos de alta y demostramos el teorema `trivialGraphsEquality`, que establece que todo grafo es observacionalmente igual a sí mismo, mientras que para demostrar que los eventos están incluídos en sí mismos fue cuestión de dar de alta y demostrar otro teorema, `equalSetsImplyInclusion`, que implica que si dos conjuntos son iguales, entonces la inclusión se implica. Utilizamos este teorema aplicandolo con `events P` terminando así la demostración.

Con la finalización de la demostración de la tercer regla, queda terminada la demostración en Coq de la correctitud de la implementación basada en estados del *RDT contador* respecto de su especificación, $\mathcal{S}_{ctr}$.

4.3.4. Consideraciones adicionales

Para el desarrollo de la demostración hemos utilizado diversos lemas que nos han permitido modularizar ciertos pasos lógicos, abstrayéndolos del problema puntual del *contador*. No hemos demostrado algunos de estos puesto que, si bien son válidos, su demostración en Coq era compleja, como es el caso de lemas como `countOfIncsOverAnUnionIsTheMax`, que indica que si `f''` es como la definimos para la regla de `receive_relation_step`, G y G' son compatibles (no tienen historia en común inconsistente) y ambos pertenecen a estados abstractos relacionados con otros estados concretos en $\mathcal{I}_{\mathcal{S}_{ctr}}$, entonces la cantidad de *incs* realizados por una réplica dada r en $G \sqcup G'$ (utilizando `f''` para identificar qué eventos fueron hechos por la réplica r) será igual al máximo de los *incs* entre G y G' realizados por la misma réplica dada r.

5. CONCLUSIONES Y TRABAJO FUTURO

Presentamos un framework en Coq que permite modelar el comportamiento especificado para un tipo de datos replicado y el de una de sus implementaciones vía los *labelled transition systems* que los representan, para luego poder enunciar el teorema que establece que la implementación del mismo es correcta respecto de su especificación y demostrarlo. Utilizamos el tipo de datos replicado *Contador* a modo de ejemplo y dimos una demostración en Coq de la correctitud de una implementación basada en estados del mismo siguiendo los lineamientos de su demostración manual presentada en [1]. El mencionado teorema fue enunciado en Coq a partir de un resultado propuesto en [1] que caracteriza la correctitud como una relación de simulación entre ambos $LTSs$. Esto nos permite abordar la demostración de correctitud de una implementación de un RDT de manera computarizada a través de las tácticas de Coq en lugar de realizarla en papel, lo cual trae importantes ventajas como una fuerte confianza en la validez de la misma y la homogeneidad de futuras demostraciones, ya que todas aquellas que usen este framework tendrán una estructura similar. Por otro lado, remarcamos que nuestro framework es fácilmente adaptable a distintos $RDTs$, basta solo contar con la especificación $\mathcal{S}$ del mismo y con el detalle de cómo se comporta la implementación frente a cada operación, es decir, su LTS concreto. Confiamos en que este framework ofrezca una alternativa más formal y mecánica a la hora de realizar demostraciones de correctitud de implementaciones de $RDTs$ especificados mediante el enfoque de especificación funcional.

En cuanto al trabajo futuro, planeamos ofrecer más demostraciones de correctitud de implementaciones de otros tipos de datos replicados a partir de: (i) el framework desarrollado en 4.1, y de (ii) llevar a cabo las formalizaciones en Coq sobre distintas implementaciones de dichos $RDTs$, tales como el *Last write wins register* [1], el *Multivalue register* [2] o el *Observed-remove set* [2].

Bibliografía

[1] Gadducci, Fabio and Melgratti, Hernán and Roldán, Christian.
On the semantics and implementation of replicated data types.

[2] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, Marek Zawirski.
Replicated Data Types: Specification, Verification, Optimality.

[3] Inria.
Coq tactics.

[4] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Brent Yorgey.
Software foundations Volume 1 - Logical Foundations.

[5] S. Gilbert and N. Lynch.
Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services.

[6] A. Lakshman and P. Malik.
Cassandra: a decentralized structured storage system.

[7] Peter Bailis and Ali Ghodsi.
Eventual consistency today: limitations, extensions, and beyond.