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

Verificación de invariantes en tipos de datos replicados con consistencia mixta

Tesis de Licenciatura en Ciencias de la Computación

Brian Ariel Bokser

Director: Christian Roldán
Codirector: Hernán Melgratti
Buenos Aires, 2019

VERIFICACIÓN DE INVARIANTES EN TIPOS DE DATOS REPLICADOS CON CONSISTENCIA MIXTA

Los sistemas geográficamente distribuidos replican su estado sobre diferentes nodos (llamados también réplicas) con el fin de satisfacer requerimientos no-funcionales como la alta disponibilidad. Sin embargo, dado que la comunicación entre réplicas es asincrónica, los usuarios de estos sistemas pueden, temporalmente, observar discrepancias en la información debido a que acceden a distintas replicas. Por lo tanto, razonar sobre la correctitud de sistemas geo-replicados se convierte en una tarea difícil. Es así como hoy en día existe la necesidad de contar con herramientas que nos permitan especificar, modelar y analizar sistemas geo-replicados. Los tipos de datos replicados han sido propuestos a tal fin. Los tipos de datos replicados son análogos a los tipos de datos abstractos, pero sus operaciones tienen en cuenta los distintos niveles de consistencia, es decir, las anomalías o inconsistencias que el sistema puede aceptar hasta que las réplicas converjan a el mismo estado.

En esta tesis abordamos el problema de probar invariantes de datos sobre tipos de datos replicados. Como modelo formal de computación de sistemas geo-replicados adoptamos QUELEA, un modelo formal de consistencia mixta, es decir, que permite combinar operaciones con diferentes niveles de consistencia. Si bien QUELEAestá equipado con un lenguaje de contratos que permite razonar sobre la consistencia de las operaciones, no provee herramientas que permitan escribir y probar invariantes de datos a nivel aplicación. Concretamente, proponemos una técnica para probar invariantes de datos sobre el framework QUELEA. Para ello, proponemos una semántica operacional alternativa que garantiza una propiedad de consistencia conocida como visibilidad causal. Luego, presentamos un conjunto de *proof-rules* que permiten verificar invariantes de datos y finalmente ilustramos como estas *proof-rules* pueden implementarse en F* , un lenguaje de programación que descarga pruebas sobre un SMT solver.

Palabras claves: Verificación, consitencia débil, sistemas distribuidos, CRDTs, teorema CAP

INVARIANT VERIFICATION FOR REPLICATED DATA TYPES WITH MIXED CONSISTENCY

Geographically distributed systems replicate their state over different nodes (called replicas) aiming at satisfying non-functional requirements such as high availability. However, given that communication between replicas is asynchronous, users of such systems may access different replicas and thus temporarily observe different information. Because of this, reasoning about the correctness of geo-replicated systems is a hard task. Then, nowadays tools that allow you to specify, model and analyze geo-replicated systems are needed. To that end, Replicated Data Types were proposed. Replicated Data Types are analogous to abstract data types, but their operations take into account different consistency levels, that is to say, anomalies or inconsistencies that the system may accept until replicas converge to the same state.

In this thesis, we address the problem of proving data integrity invariants over replicated data types. As a formal computing model of geo-replicated systems we adopt QUELEA, a model of mixed consistency, by which we mean that it allows to combine operations with different consistency levels. Even if QUELEAis equipped with a contract language that allows reasoning over operations consistency, it does not provide tools that allow you to write nor prove application invariants. Concretely, we propose a technique to prove data invariants over the QUELEAframework. To that end, we propose an alternative operational semantics that guarantees a consistency property called causal visibility. Thereafter, we present a set of proof-rules that allow verification of data invariants, and finally illustrate how these proof-rules may be implemented in $F^\star$, a programming language which relies on an SMT solver for proof automation.

Keywords: Verification, weak consistency, distributed systems, CRDTs, CAP theorem

AGRADECIMIENTOS

A todos los que componen la Facultad de Ciencias Exactas y Naturales de la Universidad de Buenos Aires por generar un hermoso espacio de aprendizaje.

Al jurado por tomarse el tiempo de leer esta tesis y aportar valiosos comentarios.

A todos mis amigos. Gracias inspirarme a ser una mejor persona.

A Hernán por la precisión de sus comentarios, que lograron darle forma a esta tesis.

A Christian, por confiar en mi y tenerme infinita paciencia. Fue un placer trabajar con vos y ver como te preocupabas no solo por lo técnico, si no también por armar un increíble vínculo de trabajo.

A mi familia por el acompañamiento y la ayuda que siempre me dieron. A mis padres por darme todas las oportunidades que me hicieron llegar hasta acá.

A Nicole por estar siempre, especialmente cuando escribía esta tesis. Sin tu apoyo este documento no sería lo que es.

A mi abuelo David

Índice general

1. INTRODUCCIÓN

Muchas de las aplicaciones utilizadas hoy en día son construidas a partir de replicar su estado en diferentes nodos. Esto permite que las aplicaciones respondan incluso cuando la red se vuelve más lenta o algún nodo de la red se cae. Sin embargo, el teorema CAP [GL02] establece que no es posible garantizar simultáneamente alta disponibilidad, tolerancia a fallas y consistencia.

Dado que la experiencia del usuario muchas veces resulta ser más importante que la consistencia, la literatura ha propuesto varios tipos de consistencia débil [VV15] donde no existe una visión única sobre el estado del sistema. En particular, en el dominio de los sistemas geo-replicados, un programa manipula datos almacenados en réplicas (servidores o agentes), a través de operaciones de lectura y escritura. Las réplicas comunican sus cambios asincrónicamente a través de una red de alta latencia, mostrando discrepancias temporales hasta eventualmente, converger a un mismo estado. Los estados observados se componen de **efectos**, que a su vez son generados por operaciones durante una ejecución de programa. Luego los efectos se transmiten a las réplicas con una garantía determinada por el nivel de consistencia elegido. Por ejemplo, la **consistencia eventual** solo garantiza que cualquier escritura será entregada a cada réplica del sistema en algún momento en el futuro.

En cambio, en consistencia fuerte existe un único estado global que los clientes pueden observar. Este es el tipo de semánticas a la que estamos acostumbrados al escribir software secuencial.

Hoy en día los sistemas geo-replicados son cada vez más comunes, con programas y aplicaciones como buscadores o redes sociales, todos ellos comunicándose sobre internet. En este contexto, sus servidores pueden encontrarse distribuidos en cualquier parte del mundo. Por ejemplo, la base de datos Dynamo de Amazon [DHJ$^+$07] es un sistema de almacenamiento de tipo *key-value* altamente disponible del cual dependen otros servicios de la empresa para lograr una experiencia de usuario "siempre disponible". Este trabajo inspiró la creación de otras bases de datos con foco en alta disponibilidad, como Cassandra [LM10] . Además algunas empresas ofrecen servicios de almacenamiento distribuido sobre los cuales los desarrolladores montan sus aplicaciones. Estos servicios cuentan con una variedad de niveles de consistencia. En el caso de Azure CosmosDB el nivel de consistencia es configurable por el desarrollador [Mic19].

Mediante el uso de bases de datos con abstracciones convenientes, el programador delega el manejo de la comunicación entre réplicas y la implementación de protocolos correspondientes. Sin embargo, la variedad en las formas de consistencia disponibles [Ter13] resulta en nuevas variables a tener en cuenta al diseñar sistemas. Asimismo, razonar sobre programas geo-distribuidos es tarea difícil, y por ende los *bugs* son frecuentes [FZWK17]. En estos programas nos encontramos con fenómenos que no encontramos en programas secuenciales. La intuición que desarrollamos a la hora de aprender a programar debe ser revisada ante la complejidad de estos sistemas. En lo que sigue, utilizaremos la palabra

anomalía para referirnos a comportamientos inesperados de una ejecución.

Para desarrollar programas de estas características hacen falta herramientas específicas. Estudiaremos una de ellas, llamada QUELEA(pronunciada"Kuilía"), un *framework* para desarrollar programas con consistencia débil sobre sistemas distribuidos, acompañado de modelos de razonamiento para probar propiedades. El mismo fue desarrollado por Sivaramakrishnan et al. [SKJ15] y fue implementado como una extensión que opera sobre el lenguaje de programación Haskell. Además permite interactuar con la base de datos Cassandra mediante una capa intermedia para proveer almacenamiento.

La componente esencial de QUELEAes el lenguaje de *contratos*. Este permite especificar propiedades de consistencia que nuestra aplicación debe cumplir utilizando fórmulas lógicas.

Además existe una semántica operacional que modela el comportamiento de programas geo-replicados construidos mediante el *framework* QUELEA.

Recientemente, han surgido herramientas para garantizar el cumplimiento de un invariante en sistemas geo-replicados [GYF+16, BND+14]. Estas herramientas operan mediante *proof-rules*, reglas que nos permiten inferir el cumplimiento de un invariante de integridad.

1.1. Contribución

En esta tesis presentamos *proof-rules* que permiten verificar el cumplimiento de un invariante de integridad en la semántica operacional de QUELEA. Las mismas están inspiradas en el desarrollo de [GYF+16] y permiten probar que un invariante se respeta en operaciones con distintos niveles de consistencia.

Además utilizamos el lenguaje de programación F* para implementar estas *proof-rules*, descargando la demostración de las mismas a un SMT solver.

1.2. Organización de la tesis

En el capítulo 2 estudiamos detalladamente el *framework* de desarrollo QUELEA. En nuestro análisis tienen lugar las características relevantes del mismo, desde el modelo teórico hasta su implementación, pasando por el lenguaje de contratos.

La semántica operacional descripta por los investigadores que crearon QUELEAes abordada a fondo en el capítulo 3. A lo largo del mismo estudiaremos la sintaxis y el modelo que ésta semántica describe permitiéndonos llegar a la conclusión de que la semántica no cumple con propiedades importantes. Mostramos ejemplos respaldando nuestros argumentos.

En el mismo capítulo presentamos una semántica alternativa que soluciona los problemas de la original, y clarifica la sintaxis.

Más aún, en el capitulo 4 abordamos las condiciones necesarias para la convergencia de estados abstractos en esta semántica, relacionando el modelo con los tipos de datos CRDTs [SPBZ11]. Este análisis nos dará un modelo mental más claro para la construcción de programas que usan QUELEA.

En el capítulo 5 analizamos el problema de la verificación de invariantes de datos sobre programas basados en esta semántica. Construimos herramientas teóricas que proveen garantías de correctitud del análisis mencionado.

Por último en el capitulo 6 construimos un modelo de un programa geo-replicado y lo verificamos, combinando las herramientas teóricas con un lenguaje de programación que descarga demostraciones sobre un *SMT solver*.

2. QUELEA

Motivados por la dificultad de crear, entender y analizar programas geo-replicados, en este capítulo presentaremos el framework Quelea. Nos basaremos en la explicación de sus autores en el reporte presentado [SKJ15]. Sin embargo, en secciones donde nos sea conveniente cambiar la notación o diferir en algún aspecto del original, lo haremos saber.

Necesitamos herramientas que nos ayuden a comprender programas con consistencias débiles, que son de uso cada vez más frecuente. Por eso en la sección 2.1 describimos el modelo, una simplificación de la situación al desarrollar sistemas geo-replicados. Posteriormente, en la sección 2.2 introducimos un ejemplo de uso del framework. Luego formalizamos la noción de ejecuciones de programas en la sección 2.3. Dada esta formalización, tenemos un dominio claro para introducir los contratos en la sección 2.4. En la sección 2.5 discutimos como axiomatizar mediante contratos, niveles de consistencia y en la sección 2.6 revisitaremos una notación para comparar contratos. Utilizando esta forma de comparación desarrollamos un método de clasificación de contratos. Finalmente, en 2.7 mencionamos las técnicas utilizadas para implementar QUELEA.

2.1. El modelo

El modelo asume la existencia de un único *store*, compuesto por una colección de réplicas identificadas como r_1, r_2, etc. Los clientes interactúan con el store mediante una serie de *operaciones* identificadas por `op`. Las operaciones generan *efectos*, denotados con la letra η, unidades que cambian el estado del programa y son transmitidas eventual y asincrónicamente a otras réplicas. El estado de una réplica es el conjunto de efectos almacenados en la misma.

La secuencia de operaciones invocadas por un cliente particular forma una *sesión* denotada con la letra σ. De ahora en más usaremos las palabras cliente y sesión de forma intercambiable para referirnos al mismo concepto. Es decir que un cliente se relacionará con la única sesión que ejecuta. Típicamente un *store* es accedido por muchos clientes (y por ende sesiones) concurrentemente.

Cuando un cliente invoca una operación, el sistema elige una réplica disponible. La especificación de la operación indica (I) cómo obtener un resultado (II) cómo generar un nuevo efecto a partir del estado de una réplica. El resultado es recibido por el cliente, y el efecto se agrega al conjunto de efectos presentes en la réplica. Luego el efecto se envía asincrónicamente a otras réplicas.

Al ejecutarse una operación en una réplica, solo algunos efectos estarán presentes en la misma. Este fenómeno lleva a que las operaciones tengan una visión parcial del estado global del sistema. Luego los efectos generados comparten esta visión parcial. Decimos que un efecto η_1 es *visible* a otro efecto η_2 si η_1 estaba presente en la réplica al momento de crearse η_2. Esta noción de cuando un efecto es visible a otro se captura con una relación llamada `vis` que es (I) irreflexiva y (II) asimétrica.

(I) $\forall \eta.\ \neg \texttt{vis}(\eta, \eta)$

(II) $\forall \eta_1 \eta_2.\ \texttt{vis}(\eta_1, \eta_2) \implies \neg \texttt{vis}(\eta_2, \eta_1)$

.

Si dos efectos son generados en la misma sesión, están relacionados mediante el orden de sesión (`so`, por *session order*).

Además el modelo admite extender el análisis a situaciones en las que una réplica almacene más de un objeto replicado. Cada efecto de una réplica pasa a aplicarse sobre un objeto en particular. Para esto se introduce la relación `sameobj`, donde dos efectos están relacionados por `sameobj` si modifican el mismo objeto. Sin embargo, en este trabajo vamos a analizar situaciones con un único objeto en las réplicas, así que todos dos efectos estarán relacionados mediante `sameobj`.

El modelo admite las inconsistencias asociadas con consistencia eventual. Mediante el uso de contratos, el programador puede identificar niveles de consistencia precisos para cada operación, de forma tal que los requerimientos de la aplicación no se violen.

2.2. Ejemplo motivacional

Considere el siguiente programa escrito mediante el framework Quelea para un torneo virtual. Los jugadores pueden inscribirse o darse de baja del torneo. Además, la cantidad de jugadores inscriptos en el torneo no debe superar nunca la capacidad del mismo. El ejemplo es una simplificación del usado por Balegas et al. [BND+14].

Utilizaremos Haskell para describir las operaciones del programa. Escribiremos un tipo algebraico `TournamentEff` que representará los efectos producidos por las operaciones.

```
type Player = String

data TournamentEff = Enroll Player |
                     Disenroll Player |
                     GetEnrolledCount |
                     IsEnrolled | deriving (Show, Eq)
```

Modelaremos el estado de cada réplica con listas de efectos, que denominaremos *historia* de un objeto.

Las operaciones modelan cómputos que alteran el estado del programa, y por ende pueden generar nuevos efectos que serán agregados a la réplica. La transmisión de los efectos a otras réplicas es transparente al programador.

Una operación de tipo `Operation a v` que se ejecuta sobre una réplica toma una lista de efectos y un valor auxiliar de tipo `a` y retorna una tupla compuesta por un valor de tipo `v` y un efecto opcional. Este último modela la posible falla de las operaciones. Por esto utilizamos el tipo `Maybe` de Haskell.

```
type Operation a v = [TournamentEff] -> a -> (v, Maybe
    TournamentEff)
```

La operación de `enroll` toma la historia del torneo y un jugador. Si la cantidad de inscriptos actual es menor que la capacidad, agrega un nuevo jugador. En caso contrario, la operación no altera el estado.

```
enroll :: Operation Player ()
enroll hist p = if (currentAmountOfPlayers hist < capacity )
                then ((), Just $ Enroll p)
                else ((), Nothing)
```

Por otro lado la operación `disenroll` remueve un jugador de la lista de inscriptos. No es necesario que la operación chequee si el jugador actualmente está inscripto. En cambio, siempre crea el efecto `Disenroll` para indicar que remueve un jugador.

```
disenroll :: Operation Player ()
disenroll hist p = ((), Just $ Disenroll p)
```

La solución también incluye los *observadores*, aquellas funciones puras (sin efectos sobre el estado del programa) que toman el estado de una réplica como parámetro y retornan un valor, posiblemente brindando información del estado actual. Este es el caso de la función `currentAmountOfPlayers`. La misma computa la cantidad de jugadores que actualmente están inscriptos al torneo. Las definiciones de este y otros observadores están en la sección 8.1 del anexo.

```
isEnrolled :: Operation Player Bool
isEnrolled hist p = (isEnrolledObs hist p, Just IsEnrolled)

getEnrolledCount :: Operation () Int
getEnrolledCount hist _ = (currentAmountOfPlayers hist, Just
   GetEnrolledCount)
```

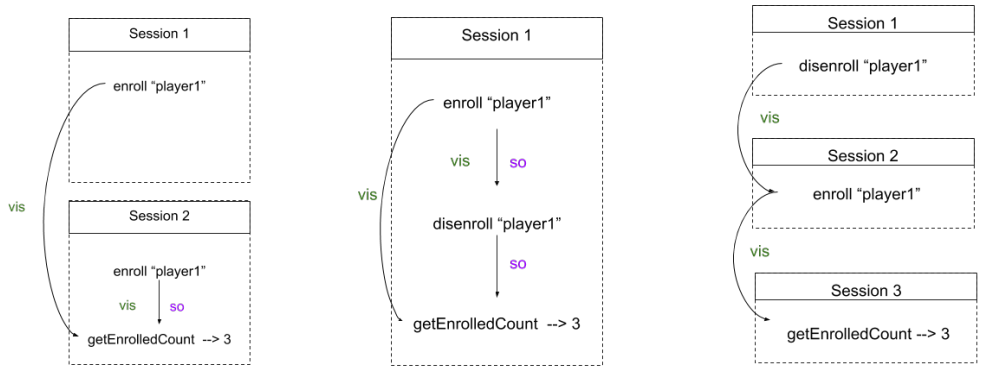
Además `isEnrolled` y `getEnrolledCount` son operaciones Quelea, y por ende tienen el tipo `Operation`. Dado que estas operaciones solo consultan el estado actual, los efectos generados no afectan el estado del objeto. Incluimos estos efectos para mayor simetría en el tratamiento de las operaciones. `isEnrolled` nos permite preguntar si un jugador está anotado en el torneo o no, mientras que `getEnrolledCount` retorna la cantidad actual de jugadores anotados.

Otros detalles de esta implementación serán explicados en secciones posteriores.

Nota 2.2.1. El ejemplo utilizado en el trabajo original [SKJ15] es distinto. Decidimos utilizar otro ejemplo debido a que algunos fenómenos más complejos no se presentaban en el original. Por ejemplo no nos permitía analizar la conmutatividad de efectos estudiada en el capítulo 4.

2.3. Ejecuciones Abstractas

El comportamiento observable de clientes interactuando con un store, es capturado abstractamente a través de una *ejecución abstracta* [BGYZ14]. Formalmente, una ejecución abstracta es una tupla $(A, vis, so, sameobj)$ definida en términos de relaciones de orden



(a) Enroll concurrente. Estado inicial: Un jugador inscripto en un torneo con capacidad máxima dos.

(b) Efecto faltante. Estado inicial: Dos jugadores inscriptos en un torneo con capacidad máxima tres.

(c) Lectura inválida. Estado inicial: dos jugadores inscriptos en un torneo con capacidad máxima dos.

Fig. 2.1: Anomalías posibles bajo consistencia eventual.

sobre un conjunto de efectos A. Al conjunto de efectos lo llamamos "sopa de efectos". Las relaciones $\mathsf{vis}, \mathsf{so}, \mathsf{sameobj} \subseteq \mathsf{A} \times \mathsf{A}$ son conocidas como relación de visibilidad, orden de sesión y mismo objeto respectivamente.

En el ejemplo de programa que vimos (sección 2.2), agregar dos jugadores al torneo concurrentemente podría llevarnos a superar la capacidad, y por ende a un estado inválido. Esta anomalía, representada en la figura 2.1a, muestra que las ejecuciones bajo consistencia eventual pueden romper invariantes del sistema.

Por otro lado, esperamos que algunas operaciones observen el resultado de lo que ocurrió antes en la sesión. En el caso del torneo, si en una sesión agregamos un jugador (`enroll`), para después chequear cuantas personas están en el torneo (`getEnrolledCount`), deseamos que el efecto resultante de la primer operación afecte la segunda. Si `getEnrolledCount` se ejecuta en una réplica donde el efecto de `enroll` no está disponible, el resultado de la operación anterior no es visible a la operación actual. Este caso está representado en la figura 2.1b

Por último, supongamos que se ejecuta una operación `disenroll`, dando lugar a que una persona más se inscriba al torneo. Este efecto es visible a una operación `enroll`. Si una operación de `getEnrolledCount` ve el efecto de `enroll` y no el efecto de `disenroll`, la cantidad de participantes que retorna excede la capacidad. La situación descripta ocurre en la figura 2.1c.

Por esto, necesitamos que las réplicas siempre mantengan estados que sean un *causal cut* (o corte causal) del conjunto de efectos. Esto quiere decir que si un estado tiene en cuenta un efecto η, también tiene que tener en cuenta sus predecesores, los efectos que estaban presentes en la réplica donde se creó η.

2.4. Contratos

Los contratos de QUELEA permiten especificar requerimientos de programas. Un contrato es una fórmula lógica que describe las propiedades que deben cumplir las ejecuciones al agregarse un efecto η. En particular, utilizan un fragmento decidible de la lógica de primer orden, donde los cuantificadores permiten versar sobre los efectos. Además, los contratos están asociados a una única operación. En la sintaxis de los contratos, $\hat{\eta}$ denota un efecto fresco, es decir, un efecto generado por la operación que el contrato tiene asociado.

Las relaciones `vis`, `so` y `sameobj` y la igualdad actúan como primitivas y pueden combinarse mediante el uso de operadores de unión, intersección y clausura transitiva para formar otras relaciones. De esta forma podemos derivar las siguientes propiedades:

- *same object session order*: $\texttt{soo} = \texttt{so} \cap \texttt{sameobj}$,
- *happens-before order*: $\texttt{hb} = (\texttt{so} \cup \texttt{vis})^+$, y
- *same object happens-before order*: $\texttt{hbo} = (\texttt{soo} \cup \texttt{vis})^+$.

En la figura 2.2 podemos ver una descripción formal del lenguaje de contratos QUELEA. El lenguaje solo admite cuantificadores universales prenexos. Estos cuantificadores pueden ser tipados o no-tipados. El tipo de un efecto es simplemente el nombre de la operación que lo generó. Los cuantificadores admiten disyunción de tipos para indicar que un efecto puede provenir de una operación u otra. En contraste, los cuantificadores no-tipados admiten efectos provenientes de cualquier operación.

$$
\begin{array}{rcll}
 & x, y, \hat{\eta} \in \texttt{EffVar} & & \mathsf{Op} \in \texttt{OperName} \\
\psi & \in \texttt{Contract} & ::= & \forall(x:\tau).\psi \mid \forall x.\psi \mid \pi \\
\tau & \in \texttt{EffType} & ::= & \mathsf{Op} \mid \tau \vee \tau \\
\pi & \in \texttt{Prop} & ::= & \mathsf{true} \mid R(x,y) \mid \pi \vee \pi \\
 & & \mid & \pi \wedge \pi \mid \pi \Rightarrow \pi \\
R & \in \texttt{Relation} & ::= & \mathsf{vis} \mid \mathsf{so} \mid \mathsf{sameobj} \mid = \\
 & & \mid & R \cup R \mid R \cap R \mid R^+
\end{array}
$$

Fig. 2.2: Lenguaje de contratos QUELEA [SKJ15]

Recordemos el ejemplo de una ejecución concurrente de dos operaciones *enroll* en la sección 2.2. Si inscribo dos jugadores concurrentemente cuando el torneo está al borde su capacidad, el estado resultante no cumple el invariante ya que la cantidad de inscriptos supera la capacidad.

El objetivo de describir propiedades a cumplir por las ejecuciones es remover los estados inválidos de las ejecuciones posibles. Es decir que al describir contratos usamos un enfoque por la positiva para prohibir estados inválidos.

En el caso del torneo, se debe garantizar que al generar un nuevo efecto de inscripción ($\hat{\eta}$), no haya otro efecto de inscripción concurrente. Es decir, que para cualquier inscripción

(e), o bien la inscripción es visible al efecto actual ($\texttt{vis}(e, \hat{\eta})$) o viceversa ($\texttt{vis}(\hat{\eta}, e)$). En el lenguaje formal contratos QUELEA escribimos:

$$\psi_{enroll} = \forall e : enroll.\ \texttt{vis}(e, \hat{\eta}) \vee \texttt{vis}(\hat{\eta}, e) \vee e = \hat{\eta}$$

En cambio, la operación *disenroll* no tiene ningún requerimiento. Es seguro ejecutarla bajo consistencia eventual. Luego, el contrato asociado es trivial:

$$\psi_{disenroll} = true$$

Por último, las operaciones *getEnrolledCount* e *isEnrolled* requieren haber visto todo lo que ocurrió en la sesión, y requieren observar un corte causal del conjunto de efectos. Esto es porque al consultar el estado actual, esperamos ver la última información, incluyendo aquellos efectos generados en operaciones anteriores de la sesión. Es decir, si un efecto está en la misma sesión debo verlo, y si veo un efecto, debo ver transitivamente aquellos que este vió. En términos de contratos, si existen efectos a, b tal que $vis(a, b)$ y $vis(b, \hat{\eta})$ entonces $vis(a, \hat{\eta})$

$$\psi_{getEnrolledCount} = \forall a, b.((\texttt{vis}(a, b) \wedge \texttt{vis}(b, \hat{\eta}) \implies \texttt{vis}(a, \hat{\eta})) \wedge (soo(c, \hat{\eta}) \implies vis(c, \hat{\eta}))$$

Las ejecuciones abstractas nos permiten interpretar el lenguaje de contratos QUELEA. De esta forma, los cuantificadores tienen un dominio claro, que es el de los efectos pertenecientes a la sopa, y las relaciones primitivas como vis se interpretan mediante su equivalente en la ejecución abstracta. Luego podemos usar las ejecuciones E como potenciales modelos de una fórmula perteneciente al lenguaje de contratos QUELEA. Si E es un modelo de una fórmula ψ escribimos $E \models \psi$.

Definición 2.4.1 (Satisfacción de contratos)**.** Decimos que una ejecución E *satisface un contrato* ψ *si y solo si* $E \models \psi$.

Por otro lado, si analizamos la definición de ejecución abstracta nos daremos cuenta que es muy permisiva. Hay ciertos fenómenos que no van a ocurrir en ninguna ejecución real de programa. Por ejemplo, ¿Qué pasa si hay ciclos en la relación de visibilidad? Recordemos que la relación de visibilidad representa que un efecto conoce a otro a la hora de crearse. De esta forma si un efecto conoce a otro, el primero se creó antes. Un ciclo en la relación de visibilidad representaría que un efecto se creo antes que si mismo.

Para evitar considerar ejecuciones abstractas imposibles (las que no tienen una interpretación en el mundo real), caracterizamos un conjunto de ejecuciones que llamamos *bien formadas*.

Nota 2.4.1. En lo que sigue hbo refiere a la relación *happens-before* introducida por Lamport. La formalización está en la sección 2.4. [Lam78]

Definición 2.4.2 (Well-Formed)**.** Una ejecución abstracta $E = (\mathsf{A}, \mathsf{vis}, \mathsf{so}, \mathsf{sameobj})$ es Well-Formed (bien-formada) si cumple el siguiente conjunto de axiomas (Δ):

- La relación happens-before es acíclica: $\forall a.\neg \texttt{hbo}(a, a)$
- Los efectos relacionados por `vis` pertenecen al mismo objeto: $\forall\ a, b.\ \texttt{vis}(a, b) \implies \texttt{sameobj}(a, b)$
- La relación de orden de sesión es transitiva: $\forall\ a, b, c.\ \texttt{so}(a, b) \wedge \texttt{so}(b, c) \implies \texttt{so}(a, c)$
- `sameobj` es una relación de equivalencia:
  - $\forall\ a.\ \texttt{sameobj}(a, a)$
  - $\forall\ a, b\ \texttt{sameobj}(a, b) \implies \texttt{sameobj}(b, a)$
  - $\forall\ a, b, c\ \texttt{sameobj}(a, b) \wedge \texttt{sameobj}(b, c) \implies \texttt{sameobj}(a, c)$

2.5. Garantías de consistencia

Hasta ahora nos concentramos en la expresabilidad de requerimientos de una aplicación en el lenguaje de contratos de QUELEA. Sin embargo, los requerimientos no siempre son triviales de cumplirse. Por lo general van a cumplirse mediante la selección de niveles de consistencia adecuados para cada operación. ¿Podemos expresar niveles de consistencia ya estudiados en la literatura como un contrato QUELEA? Las bases de datos disponibles ofrecen una variedad de niveles de consistencia que queremos modelar.

- **Eventual Consistency**.

  En [SKJ15] se utiliza intercambiadamente *eventual consistency* y *causal visibility*.

  Allí se denomina ψ_{ec} (*eventual consistency*) al contrato de consistencia que realmente refiere a **causal visibility**.

  Esta última propiedad nos dice que el store siempre muestra un subconjunto causal de los efectos. Es decir, si un efecto a ocurre antes que otro efecto b, y b es visible a $\hat{\eta}$, entonces también a debe ser visible a $\hat{\eta}$.

  $\psi_{ec} = \forall a, b\ \texttt{hbo}(a, b) \wedge \texttt{vis}(b, \hat{\eta}) \implies \texttt{vis}(a, \hat{\eta})$

  QUELEA requiere para su correcto funcionamiento que el store provea *causal visibility* como mínimo para toda operación.

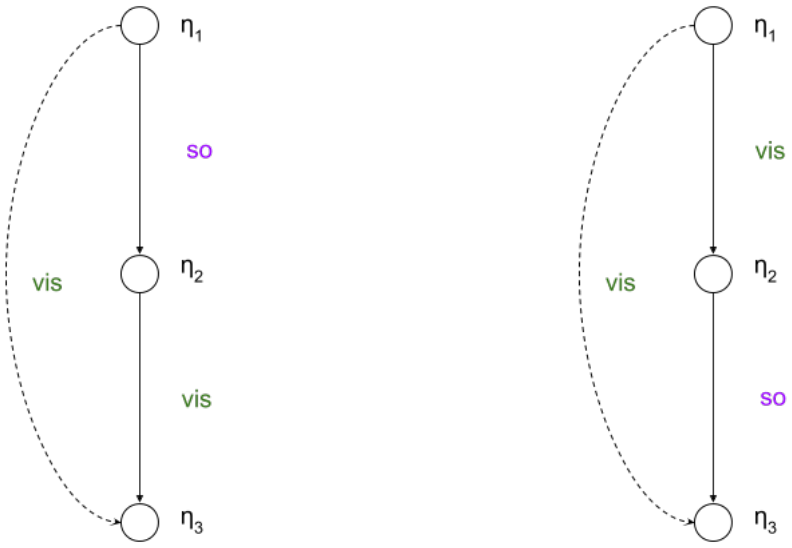
  Notar que como $\texttt{vis} \subseteq \texttt{hbo}$, este nivel de consistencia en todas nuestras operaciones implica que vis es transitivo.

- **Causal Consistency**. La consistencia causal para una operación nos indica que la relación de visibilidad debe respetar el orden *happens-before*.

  Es decir que la consistencia causal va un paso más allá y logra que dada una operación actual y su correspondiente sesión, todos los efectos generados en tal sesión sean visibles a la operación.

  Además cumple todas las propiedades que cumplía la visibilidad causal. La figura 2.3 muestra una comparación entre los dos niveles de consistencia.

  $\psi_{cc} = \forall a.\ \texttt{hbo}(a, \hat{\eta}) \implies \texttt{vis}(a, \hat{\eta})$



(a) Causal Visibility *(b)* Causal Consistency

Fig. 2.3: Comparación de dos ejecuciones abstractas que cumplen contratos de consistencia. Las flechas continuas representan hipótesis de la propiedad especificada. La flecha punteada denota la relación que es parte del consecuente. η_3 representa el efecto actual ($\hat{\eta}$). La ejecución 2.3a cumple el contrato ψ_{ec} pero no necesariamente ψ_{cc}, mientras que la ejecución 2.3b cumple ψ_{cc}

- **Strong Consistency**. La consistencia fuerte axiomatiza las nociones utilizadas por programadores desde que empiezan a escribir programas secuenciales. Este nivel de consistencia nos dice que todo efecto previamente generado es visible al efecto actual. Las operaciones con esta semántica podrían bloquearse indefinidamente, y por ende son costosas.

  $\psi_{sc} = \forall a.\texttt{sameobj}(a, \hat{\eta}) \implies \texttt{vis}(a, \hat{\eta}) \vee a = \hat{\eta} \vee \texttt{vis}(\hat{\eta}, a)$

¿Podemos razonar sobre **Eventual Consistency**? La consistencia eventual [BG13] garantiza que en algún momento todos los efectos se propagarán, y que si no hay efectos nuevos, las réplicas convergen al mismo estado. Dado que no tenemos herramientas en nuestra lógica para versar sobre la transmisión a futuro de un efecto, esta garantía será implícita.

Para poder implementar el framework QUELEA el store debe poder ofrecernos Eventual Consistency como garantía básica.

2.6. Clasificación

Ya hemos visto cómo utilizar el lenguaje de contratos QUELEA tanto para declarar los requerimientos de una aplicación como para declarar niveles de consistencia. ¿Cómo puedo

garantizar que el nivel de consistencia elegido para mi operación garantiza los requerimientos de la misma? El objetivo es determinar cuando una fórmula es suficientemente débil como para ser garantizada por un nivel de consistencia. Con esta idea en mente se define la relación de fuerza entre fórmulas.

Definición 2.6.1 (Más Débil)**.** Un contrato ψ_{op} es más débil que ψ_{st} ($\psi_{op} \leq \psi_{st}$) si y solo si $\Delta \vdash \forall \hat{\eta}.\ \psi_{st} \implies \psi_{op}$

El cuantificador ($\forall$) denota la interpretación del símbolo $\hat{\eta}$ como un efecto arbitrario. Recordemos que el contexto (Δ) en el lado izquierdo del secuente es el conjunto de axiomas base que definen cuando una ejecución es bien formada.

Si un contrato ψ_{op} de una operación (`op`) es más débil que un nivel de consistencia del store ψ_{st}, entonces los requerimientos expresados por ψ_{op} son garantizados por ψ_{st}. La completitud de la lógica de primer orden nos permite concluir que si $E \models \psi_{st}$ entonces $E \models \psi_{op}$. Luego, podemos ejecutar de forma segura la operación op bajo un nivel de consistencia ψ_{st}.

¿Es posible elegir automáticamente el nivel de consistencia adecuado para una operación? Elegir el nivel de consistencia es un problema complejo para los programadores. Si elegimos un nivel de consistencia muy débil, nos arriesgamos a incumplir invariantes de nuestra aplicación. Si elegimos un nivel de consistencia muy fuerte, la operación puede volverse demasiado costosa. De aquí que queremos un algoritmo que haga el trabajo pesado por nosotros. En esta sección tendremos en cuenta los niveles de consistencia axiomatizados por los contratos ψ_{ec}, ψ_{cc} y ψ_{sc}. Observemos que estos están totalmente ordenados bajo la relación $\leq$, siendo $\psi_{ec} \leq \psi_{cc} \leq \psi_{sc}$. Por ejemplo, si un contrato es satisfacible bajo el nivel de consistencia ψ_{ec}, entonces de seguro es satisfacible bajo ψ_{cc} y ψ_{sc}.

En la figura 2.4 definimos las reglas de clasificación que nos permiten obtener el nivel de consistencia más débil que fuerza un contrato. Notar que si un contrato no es garantizable bajo consistencia fuerte (ψ_{sc}) no es considerado un contrato bien formado (*well-formed*).

Dado que QUELEA se restringe a un fragmento decidible de la lógica, la clasificación de contratos es un problema automatizable.

Luego, la clasificación de operaciones para nuestro ejemplo da los siguientes resultados:

- EventuallyConsistent($\psi_{disenroll}$)
- CausallyConsistent($\psi_{getEnrolledCount}$)
- CausallyConsistent($\psi_{isEnrolled}$)
- StronglyConsistent(ψ_{enroll})

2.7. Implementación

QUELEA está implementado sobre GHC Haskell y funciona sobre el motor de bases de datos eventualmente consistente Cassandra. Este es responsable del manejo distribuido

$$\frac{\psi \leq \psi_{\mathsf{sc}}}{\mathsf{WellFormed}(\psi)} \qquad \frac{\psi \leq \psi_{\mathsf{ec}}}{\mathsf{EventuallyConsistent}(\psi)}$$

$$\frac{\psi \not\leq \psi_{\mathsf{ec}} \quad \psi \leq \psi_{\mathsf{cc}}}{\mathsf{CausallyConsistent}(\psi)} \qquad \frac{\psi \not\leq \psi_{\mathsf{cc}} \quad \psi \leq \psi_{\mathsf{sc}}}{\mathsf{StronglyConsistent}(\psi)}$$

Fig. 2.4: Clasificación de contratos

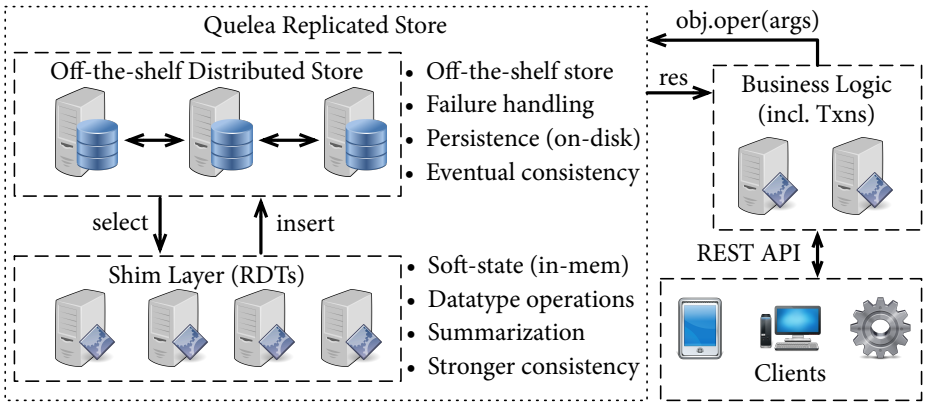


Fig. 2.5: Esquema de la implementación. Figura 10 del reporte sobre QUELEA [SKJ15].

de los datos (replicado, convergencia, *fault-tolerance* y disponibilidad). QUELEA utiliza Template Haskell para implementar la clasificación estática de contratos, descargando las demostraciones necesarias sobre Z3, el *SMT solver* (SMT son las siglas en inglés de satisfabilidad módulo teorías).

Entre el programa y Cassandra hay una capa intermedia (una *Shim layer*) que se ocupa de garantizar los niveles de consistencia. La capa intermedia mantiene en memoria una visión causalmente consistente de los objetos, llevando la cuenta de las dependencias entre efectos introducidas por las relaciones de visibilidad, sesión y mismo objeto. Además cada nodo de la capa intermedia actúa como una caché, dado que Cassandra provee los servicios de durabilidad, convergencia y tolerancia a fallas.

Hasta ahora solo hemos visto en qué consiste el modelo QUELEA y su lenguaje de declaración de contratos. Sin embargo, surgen algunas preguntas: ¿Cómo pueden implementarse lenguajes que sigan este modelo? ¿Podemos formalizar un sistema de transiciones $\xrightarrow{\eta}$ para razonar sobre estos programas?

Para responder los nuevos interrogantes, en el próximo capítulo veremos cómo proveer una semántica operacional que modela los programas geo-replicados que utilizan el framework QUELEA.

3. SEMÁNTICA OPERACIONAL

En este capítulo presentaremos la semántica operacional descripta en [SKJ15] para programas cuyos datos están replicados, y cuyas operaciones tienen distintos requerimientos de consistencia. Daremos trazas de ejemplo para entender como funciona la semántica.

A partir de un estudio detallado sobre el modelo de comportamiento de QUELEA, daremos una caraterización alternativa para el modelo operacional, ofreciendo una descripción más robusta y consistente con el propósito en el que fue desarrollado dicho framework.

3.1. Sintaxis

Cada estado de la ejecución es una tupla (E, Θ, Σ), compuesta por una ejecución abstracta E, un store Θ y un conjunto Σ de sesiones compuestas en paralelo, también llamado sopa de sesiones. La flecha $\xrightarrow{\eta}$ representa la relación de reducción entre los estados de la ejecución.

Además la semántica operacional modela el store Θ, como un diccionario de réplicas a sus estado locales. El estado local es un conjunto de efectos que la réplica ve. La garantía de consistencia mínima implementada por el store es la de visibilidad causal.

La semántica depende de la especificación de las operaciones. Esta especificación se asume global. La formalización del tipo de datos es una tupla (δ, Λ, Ψ). δ es el tipo de datos. En lo que sigue no tiene ningún uso concreto. Λ es un diccionario de identificadores de operación a una expresión. Las operaciones son escritas utilizando el *cálculo lambda*, sin dar mayores restricciones sobre su sintaxis. Las sesiones son listas finitas cuyos elementos identifican operaciones. Ψ es un diccionario de identificadores de operaciones a contratos. Recordemos que las ejecuciones abstractas E son tuplas $(\mathsf{A}, \mathsf{vis}, \mathsf{so}, \mathsf{sameobj})$, donde A es la sopa de efectos global, y $\mathsf{vis}, \mathsf{so}, \mathsf{sameobj}$ son las relaciones de visibilidad, orden de sesión y mismo objeto entre efectos de la sopa.

$s \in \texttt{SessID}$	$i \in \texttt{SeqNo}$	$r \in \texttt{ReplID}$		
η	$\in$ Effect	::=	$(s,\ i,\ op,\ v)$	
A	$\in$ A	::=	$\overline{\eta}$	
vis, so, sameobj	$\in$ Relations	::=	$\mathsf{A} \times \mathsf{A}$	
E	$\in$ ExecState	::=	(A, vis, so, sameobj)	
Θ	$\in$ Store	::=	$r \mapsto \overline{\eta}$	
τ	$\in$ ConsistencyClass	::=	ec, cc, sc	
σ	$\in$ Session	::=	$\cdot \mid op_\tau ; \sigma$	
Σ	$\in$ Session Soup	::=	$\langle s, i, \sigma \rangle \parallel \Sigma \mid \emptyset$	
	Config	::=	$(\mathsf{E}; \Theta; \Sigma)$	

(a) Gramática para estados replicados

δ	$\in$	ReplicatedDatatype		
v	$\in$	Value		
e	$\in$	Expression		
op	$\in$	Operation		
Λ	$\in$	OperationDef.	::=	$op \mapsto e$
ψ	$\in$	Contract		
Ψ	$\in$	OperationContract	::=	$op \mapsto \psi$
		RDTSpecification	::=	(δ, Λ, Ψ)

(b) Gramática para tipos de datos

Fig. 3.1: Gramática de QUELEA. Basada en la sección 5 de [SKJ15]

$$[\textsc{Oper}]\ \frac{r \in \mathsf{dom}(\Theta) \qquad \mathsf{ctxt} = \mathsf{ctxt}^*(\Theta(\mathsf{r})) \qquad \Lambda(\mathsf{op})(\mathsf{ctxt}) \rightsquigarrow^* \mathsf{v} \qquad \eta = (\mathsf{s},\ \mathsf{i},\ \mathsf{op},\ \mathsf{v}) \quad \mathsf{A}' = \{\eta\} \cup \mathsf{A} \qquad \mathtt{vis}' = \Theta(r) \times \{\eta\} \cup \mathtt{vis} \qquad \mathtt{so}' = \mathsf{A}_{(\mathsf{SessID}=s,\, \mathsf{SeqNo}<i)} \times \{\eta\} \cup \mathtt{so} \quad \mathtt{sameobj}' = \mathsf{A}' \times \mathsf{A}'}{\Theta \vdash ((\mathsf{A}, \mathtt{vis}, \mathtt{so}, \mathtt{sameobj}), \langle s, i, op\rangle) \xrightarrow{r} ((\mathsf{A}', \mathtt{vis}', \mathtt{so}', \mathtt{sameobj}'), \eta)}$$

3.2. Sistema de transiciones

A continuación comentaremos las reglas del sistema de transición, primero informalmente, y después con la sintaxis formal. Hay dos capas de reducciones que permiten darnos la semántica operacional:

- Por un lado la regla auxiliar [Oper]. Esta nos permite concluir juicios de la forma:

  $$\Theta \vdash (E, \langle s, i, \mathtt{op}\rangle) \xrightarrow{r} (E', \eta)$$

  describiendo cuando una ejecución abstracta E progresa a E' generando un efecto η en un store Θ en la réplica r. El efecto fue generado bajo la operación `op`, en la instrucción i de la sesión s. Esta información compone la tupla $\langle s, i, \mathtt{op}\rangle$).

  Para construir el nuevo efecto generado, la regla utiliza el contexto de la operación, es decir, el estado local $\Theta(r)$ sobre el cual se ejecuta una operación. La función `ctxt` nos da información relevante de los efectos para la operación. Esto es el tipo de efecto (la operación que lo generó) y un valor asociado. $\mathtt{ctxt}^*$ es una extensión de la misma función que opera sobre conjuntos de efectos.

  La definición de una operación $\Lambda(\mathtt{op})$ es un término del cálculo lambda y recibe el contexto para poder reducirse a un efecto. $\Lambda(\mathtt{op})(\mathtt{ctxt}) \rightsquigarrow^* \mathtt{v}$

  Utilizando este valor se define η, tomando en cuenta también la operación que lo generó, el identificador de sesión y la posición en la sesión.

  A continuación definimos la nueva sopa de efectos A' agregando el efecto η. Luego $\mathtt{vis}'$ agrega elementos a `vis` relacionando a η aquellos efectos que estaban presente en la réplica donde se creó. Además $\mathtt{so}'$, el session order, relaciona a η con aquellos efectos de la sopa A filtrando aquellos que pertenecen a la misma sesión y son anteriores. En símbolos $\mathtt{so}' = \mathsf{A}_{(\mathsf{SessID}=s,\, \mathsf{SeqNo}<i)} \times \{\eta\} \cup \mathtt{so}$.

- Por otro lados tenemos reglas que producen juicios de la forma:

  $$(E, \Theta, \Sigma) \xrightarrow{\eta} (E', \Theta', \Sigma')$$

  .

  Corresponden a los distintos niveles de consistencia y a la transmisión de efectos a las réplicas.

  - La regla [EffVis] es la encargada de agregar los efectos generados a las réplicas. Tiene como precondición para agregar un efecto η que aquellos otros efectos

visibles al mismo pertenezcan a la réplica. Esto hace que la réplica almacena un *causal cut* de la sopa de efectos.

Además aquellos efectos anteriores a η en la sesión también deben estar presentes en la réplica para que un efecto se agregue.

$$[\textsc{EffVis}]\frac{\eta \in \mathsf{A} \quad \eta \notin \Theta(r) \qquad E.\mathtt{vis}^{-1}(\eta) \cup E.\mathtt{so}^{-1}(\eta) \subseteq \Theta(r) \qquad \Theta' = \Theta[r \mapsto \{\eta\} \cup \Theta(r)]}{(E;\Theta;\Sigma) \xrightarrow{\eta} (E;\Theta';\Sigma)}$$

- La regla [Ec] ejecuta una operación bajo la consistencia EC, lo cual resulta en agregar un efecto a la sopa de efectos. No hay restricciones adicionales, por lo cual las ejecuciones cumplen solamente el nivel de consistencia que especifica ψ_{ec}, es decir *causal visibility* (ver capítulo 2).

$$[\text{EC}]\frac{\tau = \mathsf{EventuallyConsistent} \qquad \Theta \vdash (E, \langle s, i, op\rangle) \xrightarrow{r} (E', \eta)}{(E;\Theta;\langle s, i, op_\tau; \sigma\rangle \parallel \Sigma) \xrightarrow{\eta} (E';\Theta;\langle s, i+1, \sigma\rangle \parallel \Sigma)}$$

- La regla [Cc] ejecuta una operación bajo la consistencia CC, lo cual resulta en agregar un efecto a la sopa de efectos. Esta regla pide como requisito que todos los efectos previos en la sesión esten presentes en la réplica donde se ejecuta la operación. Esto quiere decir que el nuevo efecto η ve todos los efectos que se llevaron a cabo en la misma sesión.

  Notar que es necesario usar la regla [EffVis] incluso para propagar un efecto a la propia réplica donde se generó.

$$[\text{CC}]\frac{\Psi(op) = \mathsf{CausallyConsistent} \qquad \Theta \vdash (E, \langle s, i, op\rangle) \xrightarrow{r} (E', \eta) \qquad E.\mathtt{so}^{-1}(\eta) \subseteq \Theta(r)}{(E;\Theta;\langle s, i, op_\tau; \sigma\rangle \parallel \Sigma) \xrightarrow{\eta} (E';\Theta;\langle s, i+1, \sigma\rangle \parallel \Sigma)}$$

- La regla [ScOriginal] ejecuta una operación bajo la consistencia SC agregando un efecto a la sopa de efectos. La regla requiere que todos los efectos de la sopa estén presentes en la réplica donde se ejecutará la operación. Es decir que el nuevo efecto η verá todos los efectos ($\forall a.\ vis(a, \eta)$). Además, la regla transmite inmediatamente el efecto a todas las réplicas, agregándolo a las mismas.

$$[\textsc{SCOriginal}]\frac{\begin{array}{c}\Psi(op) = \mathsf{StronglyConsistent} \\ \Theta \vdash (E, \langle s, i, op\rangle) \xrightarrow{r} (E', \eta) \qquad E.\mathsf{A} \subseteq \Theta(r) \\ \mathsf{dom}(\Theta') = \mathsf{dom}(\Theta) \qquad \forall \mathsf{r}' \in \mathsf{dom}(\Theta').\Theta'(\mathsf{r}') = \Theta(\mathsf{r}') \cup \{\eta\}\end{array}}{(E;\Theta;\langle s, i, op_\tau; \sigma\rangle \parallel \Sigma) \xrightarrow{\eta} (E';\Theta';\langle s, i+1, \sigma\rangle \parallel \Sigma)}$$

Definición 3.2.1 (Traza)**.** En lo que sigue llamamos **traza** a elementos en la clausura transitiva de $\xrightarrow{\eta}$, es decir $t \in (\xrightarrow{\eta})^*$.

Describiremos las trazas aplicando reiteradas veces las reglas de reducción de $\xrightarrow{\eta}$.

Vamos a probar que la regla [SCOriginal] no asegura visibilidad causal mostrando una traza que la usa.

Notamos que la regla [SCOriginal] agrega un efecto η generado a todas las réplicas. Sin embargo, ¿Qué ocurre si agrega este efecto sin agregar sus anteriores bajo la relación de visibilidad? Ante este escenario, la garantía de visibilidad causal se rompe.

Ejemplo 3.2.1. *Considere el siguiente escenario. Dado el programa visto en la sección 2.2, con un torneo donde la capacidad máxima es de un solo jugador y un store Θ de dos réplicas. Dos clientes ejecutan operaciones. El primero ejecuta las operaciones* `enroll`*,* `disenroll` *y de nuevo* `enroll`*, para luego finalizar la sesión (denotado mediante $\cdot$). El otro cliente ejecuta exclusivamente* `getEnrolledCount`*.*

$$(E_0, \Theta_0, \Sigma_0) \text{ con } E_0 = \emptyset, \quad \Theta_0 = \begin{cases} r_1 \to \emptyset \\ r_2 \to \emptyset \end{cases} \quad y\ \Sigma_0 = \langle clienteA, 0, enroll\ j_1; disenroll\ j_2; enroll\ j_3; \cdot \rangle \parallel \langle clienteB, 0, getEnrolledCount; \cdot \rangle$$

Un cliente ejecuta la operación enroll mediante la regla SCOriginal *generando un efecto η_1, obteniendo así el estado $(E_1, \Theta_1, \Sigma_1)$.*

$$(E_0, \Theta_0, \Sigma_0) \xrightarrow{\eta_1} (E_1, \Theta_1, \Sigma_1) \text{ con } E_1 = (\{\eta_1\}, \emptyset, \emptyset, \emptyset), \quad \Theta_1 = \begin{cases} r_1 \to \{\eta_1\} \\ r_2 \to \{\eta_1\} \end{cases} \quad y$$
$$\Sigma_1 = \langle clienteA, 1, disenroll\ j_2; enroll\ j_3; \cdot \rangle \parallel \langle clienteB, 0, getEnrolledCount; \cdot \rangle$$

Luego, el mismo cliente ejecuta la operación disenroll bajo el nivel de consistencia EC*.*

$$\xrightarrow{\eta_2} (E_2, \Theta_2, \Sigma_2) \text{ con } E_2 = (\{\eta_1, \eta_2\}, \mathtt{vis}'', \mathtt{so}'', \mathtt{sameobj}''), \quad \Theta_2 = \begin{cases} r_1 \to \{\eta_1\} \\ r_2 \to \{\eta_1\} \end{cases} \quad y$$
$$\Sigma_2 = \langle clienteA, 2, enroll\ j_3; \cdot \rangle \parallel \langle clienteB, 0, getEnrolledCount; \cdot \rangle$$

El efecto η_2 se transmite a la réplica r_1, mediante la regla EffVis*. Notar que la cantidad de jugadores inscriptos en el estado observado por la réplica r_1 baja a cero.*

$$\xrightarrow{\eta_2} (E_3, \Theta_3, \Sigma_3) \text{ con } E_3 = (\{\eta_1, \eta_2\}, \mathtt{vis}_3, \mathtt{so}_3, \mathtt{sameobj}_3), \quad \Theta_3 = \begin{cases} r_1 \to \{\eta_1, \eta_2\} \\ r_2 \to \{\eta_1\} \end{cases} \quad y$$
$$\Sigma_3 = \langle clienteA, 2, enroll\ j_3; \cdot \rangle \parallel \langle clienteB, 0, getEnrolledCount; \cdot \rangle$$

Ahí el mismo cliente vuelve a ejecutar la operación enroll, desde la réplica r_1. La operación genera un efecto enroll dado que la cantidad actual de jugadores era menor que la capacidad.

$$\xrightarrow{\eta_3} (E_4, \Theta_4, \Sigma_4) \text{ con } E_4 = (\{\eta_1, \eta_2, \eta_3\}, \mathtt{vis}_4, \mathtt{so}_4, \mathtt{sameobj}_4), \quad \Theta_4 = \begin{cases} r_1 \to \{\eta_1, \eta_2, \eta_3\} \\ r_2 \to \{\eta_1, \eta_3\} \end{cases} \quad y$$
$$\Sigma_4 = \langle clienteB, 0, getEnrolledCount; \cdot \rangle$$

En la réplica r_2 el estado no cumple el invariante, pues la cantidad de participantes inscriptos es 2, que es mayor a la capacidad. Esto se debe a que la réplica r_2 recibió solamente los dos efectos enroll que se transmitieron instantáneamente a todas las réplicas.

Si el segundo cliente ejecuta la operación getEnrolledCount en la réplica r_2, obtendrá un conteo mayor que la capacidad.

$$\xrightarrow{\eta_4} (E_5, \Theta_5, \Sigma_5) \text{ con } E_5 = (\{\eta_1, \eta_2, \eta_3, \eta_4\}, \mathtt{vis}_5, \mathtt{so}_5, \mathtt{sameobj}_5), \quad \Theta_5 = \begin{cases} r_1 \to \{\eta_1, \eta_2, \eta_3\} \\ r_2 \to \{\eta_1, \eta_3\} \end{cases} \quad y$$
$$\Sigma_5 = \langle clienteB, 1, \cdot \rangle$$

3.3. Semántica alternativa

En esta sección describiremos aquellas modificaciones sobre la semántica operacional de la sección anterior con el fin de resolver el problema que evidenciamos con la traza anterior.

Para esto reemplazamos la regla [SCOriginal] por dos reglas con una semántica similar: [SC] y [SVC].

Por otro lado simplificaremos ligeramente la sintaxis. Todos estos cambios se ven representados en la figura 3.2. Los cambios en la gramática están señalados en rojo.

Eliminamos los contratos por operación. En su lugar, cada operación esta asociada directamente con un único nivel de consistencia.

Además cada sesión contiene una lista de operaciones. La especificación del tipo de datos contiene un diccionario (Ψ), relacionando operaciones con su nivel de consistencia.

Removemos el símbolo δ, que representaba el tipo de datos completo. Este no estaba cumpliendo ningún rol en la semántica operacional planteada.

La regla [SC] imposibilita la concurrencia del efecto generado con cualquier otro. La forma en que opera requiere que todos los efectos de la sopa estén presentes en la réplica que ejecuta la operación, como lo hacía la regla [SCOriginal]. Además la regla aplica la propagación de efectos de forma directa, igualando el conjunto de efectos de toda réplica con la sopa de efectos. Todo efecto debe propagarse a toda réplica.

Es claro que la ejecución de una implementación de la regla [SC] es costosa en términos de propagación de efectos y sus tiempos asociados. Pero este resultado es sensato, ya que son los costos asociados con la consistencia fuerte.

$$
\begin{array}{llll}
s \in \texttt{SessID} & i \in \texttt{SeqNo} & r \in \texttt{ReplID} & \\
\eta & \in \texttt{Effect} & ::= & (s,\ i,\ op,\ v) \\
\mathsf{A} & \in \mathsf{A} & ::= & \overline{\eta} \\
\mathsf{vis}, \mathsf{so}, \mathsf{sameobj} & \in \texttt{Relations} & ::= & \mathsf{A} \times \mathsf{A} \\
\mathsf{E} & \in \texttt{ExecState} & ::= & (\mathsf{A},\ \mathsf{vis},\ \mathsf{so},\ \mathsf{sameobj}) \\
 & & & \\
\Theta & \in \texttt{Store} & ::= & r \mapsto \overline{\eta} \\
\tau & \in \texttt{ConsistencyClass} & ::= & \mathsf{ec}, \mathsf{cc}, \mathsf{sc}, \mathsf{svc} \\
\sigma & \in \texttt{Session} & ::= & \cdot \mid op; \sigma \\
\Sigma & \in \texttt{Session Soup} & ::= & \langle s, i, \sigma \rangle \parallel \Sigma \mid \emptyset \\
 & \texttt{Config} & ::= & (\mathsf{E}; \Theta; \Sigma)
\end{array}
$$

(a) Gramática para estados replicados

$$
\begin{array}{llll}
v & \in \texttt{Value} & & \\
e & \in \texttt{Expression} & & \\
op & \in \texttt{Operation} & & \\
 & & & \\
\Lambda & \in \texttt{OperationDef.} & ::= & op \mapsto e \\
\Psi & \in \texttt{OperationConsistency} & ::= & op \mapsto \tau \\
 & \texttt{RDTSpecification} & ::= & (\Lambda, \Psi)
\end{array}
$$

(b) Gramática para tipos de datos

$$
[\textsc{Oper}]\ \frac{\begin{array}{c} r \in \mathsf{dom}(\Theta) \qquad \mathsf{ctxt} = \mathsf{ctxt}^*(\Theta(\mathsf{r})) \qquad \Lambda(\mathsf{op})(\mathsf{ctxt}) \rightsquigarrow^* \mathsf{v} \qquad \eta = (\mathsf{s},\ \mathsf{i},\ \mathsf{op},\ \mathsf{v}) \\ \mathsf{A}' = \{\eta\} \cup \mathsf{A} \qquad \mathtt{vis}' = \Theta(r) \times \{\eta\} \cup \mathtt{vis} \qquad \mathtt{so}' = \mathsf{A}_{(\mathsf{SessID}=s,\, \mathsf{SeqNo}<i)} \times \{\eta\} \cup \mathtt{so} \\ \mathtt{sameobj}' = \mathsf{A}' \times \mathsf{A}' \end{array}}{\Theta \vdash ((\mathsf{A}, \mathtt{vis}, \mathtt{so}, \mathtt{sameobj}), \langle s, i, op \rangle) \xrightarrow{r} ((\mathsf{A}', \mathtt{vis}', \mathtt{so}', \mathtt{sameobj}'), \eta)}
$$

$$
[\textsc{EffVis}]\ \frac{\begin{array}{c} \eta \in \mathsf{A} \quad \eta \notin \Theta(r) \\ E.\mathtt{vis}^{-1}(\eta) \cup E.\mathtt{so}^{-1}(\eta) \subseteq \Theta(r) \qquad \Theta' = \Theta[r \mapsto \{\eta\} \cup \Theta(r)] \end{array}}{(E; \Theta; \Sigma) \xrightarrow{\eta} (E; \Theta'; \Sigma)}
$$

$$
[\mathrm{EC}]\ \frac{\begin{array}{c} \Psi(op) = \mathsf{EventuallyConsistent} \\ \Theta \vdash (E, \langle s, i, op \rangle) \xrightarrow{r} (E', \eta) \end{array}}{(E; \Theta; \langle s, i, op; \sigma \rangle \parallel \Sigma) \xrightarrow{\eta} (E'; \Theta; \langle s, i+1, \sigma \rangle \parallel \Sigma)}
$$

$$
[\mathrm{CC}]\ \frac{\begin{array}{c} \Psi(op) = \mathsf{CausallyConsistent} \\ \Theta \vdash (E, \langle s, i, op \rangle) \xrightarrow{r} (E', \eta) \qquad E.\mathtt{so}^{-1}(\eta) \subseteq \Theta(r) \end{array}}{(E; \Theta; \langle s, i, op; \sigma \rangle \parallel \Sigma) \xrightarrow{\eta} (E'; \Theta; \langle s, i+1, \sigma \rangle \parallel \Sigma)}
$$

$$
[\mathrm{SC}]\ \frac{\begin{array}{c} \Psi(op) = \mathsf{StronglyConsistent} \\ \Theta \vdash (E, \langle s, i, op \rangle) \xrightarrow{r} (E', \eta) \qquad E.\mathsf{A} \subseteq \Theta(r) \\ \mathsf{dom}(\Theta') = \mathsf{dom}(\Theta) \qquad \forall \mathsf{r}' \in \mathsf{dom}(\Theta').\Theta'(\mathsf{r}') = \mathsf{E}'.\mathsf{A} \end{array}}{(E; \Theta; \langle s, i, op; \sigma \rangle \parallel \Sigma) \xrightarrow{\eta} (E'; \Theta'; \langle s, i+1, \sigma \rangle \parallel \Sigma)}
$$

$$
[\mathrm{SVC}]\ \frac{\begin{array}{c} \Psi(op) = \mathsf{StrongViewConsistent} \\ \Theta \vdash (E, \langle s, i, op \rangle) \xrightarrow{r} (E', \eta) \qquad E.\mathsf{A} \subseteq \Theta(r) \end{array}}{(E; \Theta; \langle s, i, op; \sigma \rangle \parallel \Sigma) \xrightarrow{\eta} (E'; \Theta; \langle s, i+1, \sigma \rangle \parallel \Sigma)}
$$

Fig. 3.2: Gramática alternativa.

La regla [SVC] es una versión más débil de la regla propuesta por el trabajo original de QUELEA. Para ejecutar una operación con esta semántica, la réplica debe contener todos los efectos generados hasta ahora, es decir, debe ver todos los efectos del sistema. La propagación del efecto resultante queda libre, y será propagado mediante la regla [EffVis]. Debido a las restricciones que esta aplica, la propagación respetará la causalidad de la relación vis. De esta forma concluimos que el efecto generado η solo llegará a una réplica que ya haya visto la sopa completa.

3.4. Correctitud

Definición 3.4.1 (Store consistente con visibilidad causal)**.** Un store Θ es consistente con la visibilidad causal en una ejecución $E = (\mathsf{A}, \mathsf{vis}, \mathsf{so}, \mathsf{sameobj})$ si:

$$\forall(r \in \mathsf{dom}(\Theta)).\forall(\eta \in \Theta(\mathsf{r})).\forall(\mathsf{a} \in \mathsf{A})\ \mathtt{hbo}(\mathsf{a}, \eta) \implies \mathsf{a} \in \Theta(\mathsf{r})$$

Dado que $\mathtt{soo} = (\mathsf{so} \cap \mathtt{sameobj})$ y $\mathtt{hbo} = (\mathtt{soo} \cup \mathtt{vis})^+$.

Es decir, si un efecto pertenece al conjunto efectos de una réplica, todos los efectos anteriores también pertenecen a la réplica. Aquí entendemos *anteriores* de un efecto η, como aquellos efectos a que cumplen $\mathtt{hbo}(a, \eta)$.

3.4.1. Preservación de Visibilidad Causal

Modificamos la demostración originalmente propuesta por los autores de QUELEApara la regla [SCOriginal].

Teorema 3.4.1. *Sea E una ejecución bien formada y Θ un store consistente con la visibilidad causal bajo E. Si:*

$$(E, \Theta, \Sigma) \xrightarrow{\eta} (E', \Theta', \Sigma')$$

Entonces Θ' es consistente con la visibilidad causal bajo E'

Demostración. Por análisis de casos en la transición $\xrightarrow{\eta}$

- Caso [EffVis]:

$$(E; \Theta; \Sigma) \xrightarrow{\eta} (E; \Theta'; \Sigma)$$

Como hipótesis de la regla [EffVis] sabemos que:

$$\eta \in \mathsf{A}$$

$$\eta \notin \Theta(r)$$

$$E.\mathtt{vis}^{-1}(\eta) \cup E.\mathtt{so}^{-1}(\eta) \subseteq \Theta(r)$$

$$\Theta' = \Theta[r \mapsto \{\eta\} \cup \Theta(r)]$$

Sea $r_2 \in \mathsf{dom}(\Theta')$. $\mathsf{dom}(\Theta) = \mathsf{dom}(\Theta')$. Luego $r_2 \in \mathsf{dom}(\Theta')$.

Para toda réplica que no sea r, es decir, que no sea la que recibió el nuevo efecto, el resultado se cumple porque su contenido no cambió. Formalmente, si $r_2 \neq r$:

Sea un efecto $\eta' \in \Theta'(r_2)$. Luego $\eta' \in \Theta(r_2)$. Y como Θ cumplía la garantía de consistencia, entonces

$$\forall(a \in \mathsf{A}).\ \mathtt{hbo}(a, \eta') \implies a \in \Theta(r_2)$$

Dado que $\Theta(r_2) = \Theta'(r_2)$ entonces:

$$\forall(a \in \mathsf{A}).\ \mathtt{hbo}(a, \eta') \implies a \in \Theta'(r_2)$$

Que es lo que queríamos probar.

Luego queda ver el caso $r_2 = r$.

Sea $\eta' \in \Theta'(r)$.

Queremos ver que

$$\forall(a \in \mathsf{A})\ \mathtt{hbo}(a, \eta') \implies a \in \Theta'(r)$$

.

Si el efecto no es aquel que fue propagado ($\eta' \neq \eta$) entonces $\eta' \in \Theta(r)$ dado que debe ser uno de los efectos que ya pertenecía a la réplica. Luego como Θ cumple la propiedad de visibilidad causal, ocurre que

$$\forall(a \in \mathsf{A})\ \mathtt{hbo}(a, \eta') \implies a \in \Theta(r)$$

Y como $\Theta(r) \subseteq \Theta'(r)$

$$\forall(a \in \mathsf{A})\ \mathtt{hbo}(a, \eta') \implies a \in \Theta'(r_2)$$

Que es lo que queríamos probar

Ahora distingamos el caso más importante, donde $\eta' = \eta$:

Sea $a \in \mathsf{A}$. Si $\mathtt{hbo}(a, \eta')$ veamos que $a \in \Theta'(r)$:
O bien a es un antecesor directo de η' ($\mathtt{vis}(a, \eta)$ o $\mathtt{so}(a, \eta)$) o es un antecesor transitivo.

Si es un antecesor directo, como sabemos que

$$E.\mathtt{vis}^{-1}(\eta) \cup E.\mathtt{so}^{-1}(\eta) \subseteq \Theta(r)$$

Entonces

$$a \in \Theta(r) \subseteq \Theta'(r)$$

Si es un antecesor transitivo, entonces existe un antecesor directo η'' tal que:

$$\texttt{hbo}(a, \eta'') \land (\texttt{vis}(\eta'', \eta') \lor \texttt{so}(\eta'', \eta')$$

. Como η'' es antecesor directo, debe pertenecer a la réplica r. Por lo tanto:

$$\eta'' \in \Theta(r) \land \texttt{hbo}(a, \eta'')$$

Combinándolo con la visibilidad causal de Θ sabemos que $a \in \Theta(r)$.

- Caso [EC], [CC], [SVC]:
  Ante la acción de estas reglas, el store no se ve modificado.
  Sea una réplica r y $\eta' \in \Theta(r)$.
  Supongamos que $\eta' = \eta$. Como η todavía no se ha propagado, $\eta \notin \Theta(r)$. Absurdo .
  Luego $\eta' \neq \eta$. Veamos que

  $$\forall(a \in E'.\mathsf{A}).\ E'.\texttt{hbo}(a, \eta') \implies a \in \Theta(r)$$

  Sea $a \in E'.\mathsf{A}$. Casos:

  - $a = \eta$
    Intuitivamente, no es posible que el nuevo efecto (η) venga antes que otro efecto.
    Formalmente, como η todavía no se propagó, no es posible que valga $E'.\texttt{hbo}(a, \eta')$.
    Luego la propiedad se cumple dado que el antecedente es falso.
  - $a \neq \eta$.
    $a \in E.\mathsf{A}$ y
    Por visibilidad causal de Θ,

    $$E.\texttt{hbo}(a, \eta') \implies a \in \Theta(r)$$

    Dado que los sucesores de un efecto se mantienen igual ($E'.\texttt{hbo}(a, \eta') \iff E.\texttt{hbo}(a, \eta')$)

    $$E'.\texttt{hbo}(a, \eta') \iff E.\texttt{hbo}(a, \eta') \implies a \in \Theta(r)$$

    $$E'.\texttt{hbo}(a, \eta') \implies a \in \Theta(r)$$

    Que es lo que queríamos probar

- Caso [SC]:
  Es fácil ver que la propiedad se cumple para la regla [SC], dado que cualquiera sea el efecto de la sopa ($a \in \mathsf{A}$), pertenece al store resultante ($a \in \Theta'(r)$).

□

En este capítulo describimos una semántica operacional que nos permite entender como operan los programas bajo QUELEA. Ahora podemos hacernos la siguiente pregunta: ¿Qué programas podremos expresar? ¿Qué implementaciones serán razonables utilizando esta semántica? En el siguiente capítulo exploraremos una restricción que observamos al utilizar el framework QUELEAcon la semántica estudiada.

4. CONMUTATIVIDAD

Hemos visto ejemplos de anomalías en ejecuciones de programas distribuidos. Las anomalías previamente introducidas fueron estudiadas por los autores del framework QUELEA, y por ende el lenguaje de contratos es una respuesta a ellos. En esta sección analizaremos otras anomalías que los programas geo-replicados deben superar. En particular, aquellas que tienen que ver con la **convergencia** de las réplicas hacia el mismo estado.

4.1. Convergencia

En lo que sigue, asumimos que un observador es una función que toma la *historia* de un objeto y retorna un valor. Es decir que toma una lista de efectos y nos devuelve información del estado del objeto sin comunicación con otras réplicas.

Dado que las listas que toman observadores y operaciones como *historias* de un objeto provienen del conjunto de efectos de una réplica, asumimos que estos conjuntos se codifican en listas mediante cualquier orden que respete la visibilidad causal. Es decir un orden topológico respecto a las aristas de `vis`.

Definición 4.1.1. Convergencia

Decimos que un estado (E, Θ, Σ) convergió respecto a un observador `obs` si

1. $\forall r_1, r_2 \in dom(\Theta).\ \Theta(r_1) = \Theta(r_2)$
2. Dado $r \in dom(\Theta)$, si l_1 y l_2 son órdenes topológico de $\Theta(r)$ respetando $E.\texttt{vis}$, entonces $\texttt{obs}(l_1) = \texttt{obs}(l_2)$

Esta noción de convergencia está relacionada con el concepto que Shapiro et al. llama *Strong Eventual Consistency* [SPBZ11][GKMB17]. Esto es una garantía de que no solo se propagaran eventualmente los efectos, si no que además, cuando se propaguen, el estado abstracto será igual.

Para terminar el análisis de que el fenómeno de observadores no convergentes es posible en el framework QUELEA, mostraremos una traza de la semántica operacional. Suponemos a continuación que las operaciones reciben efectos en el orden en el que estuvieron disponibles en la réplica. Este orden es un orden topológico de los efectos que respeta `vis`.

Interpretaremos los conjuntos de efectos asociados a una réplica como conjuntos ordenados. De esta forma $\{\eta_1, \eta_2\}$ es un conjunto donde η_2 es el último efecto. Es decir, estuvo disponible desde un momento posterior a η_1.

A lo largo de esta sección volveremos a trabajar el ejemplo de un programa que administra la inscripción de jugadores a un torneo (ver sección 2.2).

En el siguiente ejemplo describimos una ejecución donde dos réplicas reciben los efectos en órdenes inversos.

Ejemplo 4.1.1. *Dos sesiones interactuando con un store de dos réplicas. Los efectos no están relacionados y aparecen en las réplicas en órdenes distintos.*

$$(E_0, \Theta_0, \Sigma_0) \text{ con } E_0 = \emptyset, \quad \Theta_0 = \begin{cases} r_1 \to \emptyset \\ r_2 \to \emptyset \end{cases} \quad y\ \Sigma_0 = \langle clienteA, 0, enroll\ j_1; \cdot \rangle \parallel \langle clienteB, 0, disenroll; \cdot \rangle$$

El primer cliente ejecuta la operación enroll con el nivel de consistencia SVC.

$$\xrightarrow{\eta_1} (E_1, \Theta_1, \Sigma_1) \text{ con } E_1 = (\{\eta_1\}, \emptyset, \emptyset, \emptyset), \quad \Theta_1 = \begin{cases} r_1 \to \emptyset \\ r_2 \to \emptyset \end{cases} \quad y \quad \Sigma_1 = \langle clienteA, 1, \cdot \rangle \parallel \langle clienteB, 0, disenroll; \cdot \rangle$$

Luego la regla EffVis transmite el efecto a la réplica 1.

$$\xrightarrow{\eta_1} (E_2, \Theta_2, \Sigma_2) \text{ con } E_2 = (\{\eta_1\}, \emptyset, \emptyset, \emptyset), \quad \Theta_2 = \begin{cases} r_1 \to \{\eta_1\} \\ r_2 \to \emptyset \end{cases} \quad y \quad \Sigma_2 = \langle clienteA, 1, \cdot \rangle \parallel \langle clienteA, 0, disenroll; \cdot \rangle$$

A continuación ejecutamos la operación disenroll, con el nivel de consistencia EC

$$\xrightarrow{\eta_2} (E_3, \Theta_3, \Sigma_3) \text{ con } E_3 = (\{\eta_1, \eta_2\}, \emptyset, \emptyset, E_3.A \times E_3.A), \quad \Theta_3 = \begin{cases} r_1 \to \{\eta_1\} \\ r_2 \to \emptyset \end{cases} \quad y \quad \Sigma_3 = \langle clienteA, 1, \cdot \rangle \parallel \langle clienteB, 1, \cdot \rangle$$

Transmitimos el efecto η_2 a la réplica 2.

$$\xrightarrow{\eta_2} (E_4, \Theta_4, \Sigma_4) \text{ con } E_4 = (\{\eta_1, \eta_2\}, \emptyset, \emptyset, E_4.A \times E_4.A), \quad \Theta_4 = \begin{cases} r_1 \to \{\eta_1\} \\ r_2 \to \{\eta_2\} \end{cases} \quad y \quad \Sigma_4 = \langle clienteA, 1, \cdot \rangle \parallel \langle clienteB, 1, \cdot \rangle$$

Transmitimos el efecto η_1 a la réplica 2.

$$\xrightarrow{\eta_1} (E_5, \Theta_5, \Sigma_5) \text{ con } E_5 = (\{\eta_1, \eta_2\}, \emptyset, \emptyset, E_5.A \times E_5.A), \quad \Theta_5 = \begin{cases} r_1 \to \{\eta_1\} \\ r_2 \to \{\eta_2, \eta_1\} \end{cases} \quad y \quad \Sigma_4 = \langle clienteA, 1, \cdot \rangle \parallel \langle clienteB, 1, \cdot \rangle$$

Transmitimos el efecto η_2 a la réplica 1.

$$\xrightarrow{\eta_2} (E_6, \Theta_6, \Sigma_6) \text{ con } E_6 = (\{\eta_1, \eta_2\}, \emptyset, \emptyset, E_6.A \times E_6.A), \quad \Theta_6 = \begin{cases} r_1 \to \{\eta_1, \eta_2\} \\ r_2 \to \{\eta_2, \eta_1\} \end{cases} \quad y$$
$$\Sigma_4 = \langle clienteA, 1, \cdot \rangle \| \langle clienteB, 1, \cdot \rangle$$

4.2. Conmutatividad

Para evitar este fenómeno es suficiente probar que toda lista de efectos conmuta bajo todo observador. En otras palabras, podemos pensar en $\equiv$, una relación de congruencia entre lista de efectos, donde dos permutaciones de una lista estarían relacionadas. Los observadores deben respetar esta congruencia.

Por ejemplo consideremos el observador que determina si un jugador pertenece a los inscriptos actuales, definido de esta forma:

```
isEnrolledObsBroken [] p = False
isEnrolledObsBroken (h:hs) p = case h of
              Leave p -> False
              Enroll p -> True
              _ -> isEnrolledObsBroken hs p
```

Aquí, las siguientes dos expresiones son distintas. Por ende, el observador **no conmuta**.

`isEnrolledObsBroken ([Enroll "jugador1", Leave "jugador1"]) "jugador1"` $\to^*$ `true`

`isEnrolledObsBroken ([Leave "jugador1", Enroll "jugador1"]) "jugador1".` $\to^*$ `false`

En concreto, supongamos que tenemos una situación equivalente a la retratada por el ejemplo 4.1.1. Es decir, una ejecución con dos réplicas del programa para administrar torneos. A una de las réplicas le llegan los efectos `Enroll "jugador1"` y `Leave "jugador1"` en ese orden. A la otra llegan el orden inverso. Esto hace que el estado observado sea distinto para ambas réplicas.

Al propagarse todos los efectos a las distintas réplicas, los estados de las mismas deberían converger. Sin embargo, dan resultados distintos ante la operación `isEnrolled`.

Otra forma de escribir el observador de forma que conmute es la siguiente:

```
wasDisenrolled :: [TournamentEff] -> Set Player
wasDisenrolled hist = fromList [ p2 | Disenroll p2 <- hist]

wasEnrolled :: [TournamentEff] -> Set Player
wasEnrolled hist = fromList [ p2 | Enroll p2 <- hist]

allEnrolled :: [TournamentEff] -> Set Player
allEnrolled hist = (wasEnrolled hist \\ wasDisenrolled hist)

isEnrolledObs :: [TournamentEff] -> Player -> Bool
isEnrolledObs hist p = elem p (allEnrolled hist)
```

El observador chequea si el jugador pertenece al conjunto de inscriptos actuales. Este último esta conformado por aquellos que se inscribieron (`wasEnrolled`) pero nunca fueron removidos (`wasDisenrolled`). Se computa mediante la diferencia de conjuntos. El diseño está basado en el tipo de datos 2PSet [SPBZ11].

Notar que la conmutatividad de todos los efectos es una posible solución, suficiente para garantizar el correcto funcionamiento de los observadores, pero no necesaria. Para ser más concretos, analicemos el caso del efecto `Enroll`. Dado que este efecto solo se genera en una operación de consistencia fuerte, sabemos que dos efectos `Enroll` estarán bajo la relación de visibilidad. Por ende, no necesitamos que dos efectos `Enroll` conmuten entre ellos, pero si necesitamos que conmuten con otros efectos.

Para simplificar el análisis utilizamos simplemente la conmutatividad de todos los efectos, porque creemos que es una garantía que puede ser verificada fácilmente, tanto manual como automática. Trabajos futuros pueden abordar requerimientos más complejos.

4.3. Conflict-Free Replicated Data Type

La idea de utilizar tipos de datos cuya semántica garantice conmutatividad no es nueva. Este grupo de estructuras de datos fue estudiado formalmente en 2011 por Shapiro et al. Son llamados CRDTs [SPBZ11].

Un CRDT, (por las siglas en inglés Conflict-Free Replicated Data Type) es un tipo de datos que nos garantiza:

- Cada operación puede ejecutarse sin coordinación entre las réplicas.
- Si dos réplicas reciben las mismas actualizaciones, entonces llegan al mismo estado determinísticamente.

Para esto contamos con algunas propiedades matemáticas que el tipo de datos debe cumplir para llamarse CRDT.

Hay dos enfoques para especificar CRDTs. O bien los estados del tipo de datos conforma un *join semilattice*, o las operaciones del mismo **conmutan**. El primer enfoque corresponde a los *state-based* CRDTs y el segundo a los *op-based* CRDTs.

En un *state-based* CRDT, una operación actualiza el valor de la estructura localmente para luego transmitir su estado hacia otras réplicas. Se transmite el estado entero y por consiguiente debe implementarse una operación *merge* que condense dos estados $s1$ y $s2$. Para esto computa el *least upper bound (LUB)* de los estados.

De esta forma la transmisión de efectos cuenta con el respaldo teórico de un *join semilattice*, y por ende la función *merge* es conmutativa, asociativa e idempotente. Dadas estas propiedades podemos implementar un *state-based* CRDT utilizando un canal de transmisión con pocas propiedades. Los mensajes pueden perderse, pueden recibirse fuera de orden, o recibirse múltiples veces sin generar un problema.

En cambio un *op-based* CRDT ejecuta una operación localmente y luego transmite sus efectos secundarios individualmente a otras réplicas. Por esto, requerimos que la recepción de efectos u operaciones sea conmutativa.

Para que las réplicas alcancen la convergencia en este escenario, debemos utilizar una implementación de "reliable broadcast" que nos garantice la recepción de cada operación exactamente una vez. De esta forma los mensajes no se pierden y todas las réplicas reciben todas las operaciones. Es así que tenemos más requisitos sobre el canal de transmisión que al implementar un *state-based* CRDT.

Sin embargo el grupo de Shapiro muestra que siempre es posible emular el comportamiento de un *state-based* CRDT utilizando un *op-based* CRDT y viceversa [SPBZ11].

Por todo esto, al utilizar CRDTs para describir tipos de datos en el framework QUELEA, tendremos la garantía de que el sistema converge eventualmente a un mismo valor en todas las réplicas.

5. VERIFICACIÓN

A lo largo de este capítulo intentaremos responder la siguiente pregunta en el contexto del framework QUELEA. ¿Podemos garantizar que la ejecución de operaciones es segura? Diremos que una ejecución es segura si cumple los invariantes definidos por el programador la aplicación.

El objetivo será desarrollar teorías que nos permitan asegurar la correctitud de nuestros programas en entornos geo-distribuidos.

En la sección 5.1 daremos definiciones formales que dejan en claro los objetos matemáticos que nos servirán de base. Luego en 5.2 definiremos qué es un invariante, y cuándo un programa lo cumple. Por último, en la sección 5.3 demostraremos la correctitud de un método de verificación de invariantes vía *proof-rules*.

5.1. Formalización

Recordemos que una especificación (Λ, Ψ) es una tupla de diccionarios que dado un identificador de una operación $\mathtt{op}$, nos permite encontrar su definición y su nivel de consistencia. Estamos interesados en considerar solo aquellos estados (E, Θ, Σ) que pueden ocurrir en nuestro sistema. ¿Cómo los caracterizamos?

Definición 5.1.1 (Estado alcanzable)**.** Decimos que un estado (E, Θ, Σ) es alcanzable si existe un Σ' tal que $(\emptyset, \emptyset, \Sigma') \rightarrow^* (E, \Theta, \Sigma)$.

Además vamos a modelar aspectos de la semántica operacional asociada nuestros programas. Utilizaremos conjuntos de efectos que respeten la relación de visibilidad. De esta forma modelaremos los conjuntos de efectos que contiene cada réplica.

Definición 5.1.2 (Conjunto Causally Consistent)**.** Dado una ejecución abstracta $E = (\mathsf{A}, \mathsf{vis}, \mathsf{so}, \mathsf{sameobj})$, decimos que un conjunto de efectos $\mathcal{S}$ es Causally Consistent (escrito $CC(\mathcal{S})$) si y solo si $\mathcal{S}$ es *downward closed* respecto a $\mathtt{vis}$. Es decir, $\forall \eta.\ \eta \in \mathcal{S} \implies \mathtt{vis}^{-1}(\eta) \subseteq \mathcal{S}$

La siguiente notación nos permitirá caracterizar mejor los conjuntos de efectos sobre las que queremos versar y probar propiedades. No cualquier conjunto es aceptable en el framework QUELEA.

Notación 5.1.1. Escribimos $\mathcal{S} \sqsubseteq E$ si y solo si $\mathcal{S} \subseteq E.\mathsf{A} \wedge CC(\mathcal{S})$.

Definimos una relación de reducción auxiliar que servirá para definir otros conceptos más fácilmente. La regla $\rightarrow^{\eta}_{\mathtt{op},\tau}$ modela la ejecución de una operación $\mathtt{op}$ combinado con el momento de creación del efecto asociado. La única regla asociada aparece en la figura 5.1.

$$\frac{(E;\Theta;\langle s,i,op;\sigma\rangle \parallel \Sigma) \xrightarrow{\eta} (E';\Theta;\langle s,i+1,\sigma\rangle \parallel \Sigma) \quad \Psi(op)=\tau}{(E;\Theta;\langle s,i,op;\sigma\rangle \parallel \Sigma) \rightarrow^{\eta}_{\mathsf{op},\tau} (E';\Theta;\langle s,i+1,\sigma\rangle \parallel \Sigma)}$$

Fig. 5.1: Reglas de reducción $\rightarrow^{\eta}_{\mathsf{op},\tau}$

5.2. Preservación de invariantes

Nuestra preocupación inicial constaba de verificar que una ejecución es segura. Para caracterizar formalmente ejecuciones seguras, definiremos la noción de invariantes.

Definición 5.2.1 (Invariante)**.** Un invariante es una función que toma un conjunto de efectos y retorna un booleano, $\mathcal{I} : \text{Effect}^* \rightarrow \text{Bool}$.

El siguiente teorema nos garantiza que es suficiente verificar $\mathcal{S} \sqsubseteq E.\mathcal{I}(\mathcal{S})$ para saber que cada réplica estará en un estado que cumple el invariante.

Teorema 5.2.1 (Preservación de invariantes storeless)**.** *Sea* (E, Θ, Σ) *un estado alcanzable. Luego 1.* Θ *es Causally Consistent y, 2. Sea* $\mathcal{I} : \mathit{Effect}^* \rightarrow \mathit{Bool}$*. Si* $\forall \mathcal{S} \sqsubseteq E.\mathcal{I}(\mathcal{S})$ *entonces* $\forall r \in \mathsf{dom}(\Theta).\mathcal{I}(\Theta(\mathsf{r}))$

Demostración.

1. El store vacío ($\emptyset$ en toda réplica) es causally consistent. Como consecuencia del teorema 3.4.1 y aplicando el resultado transitivamente, Θ también es causally consistent.

2. Sea (E, Θ, Σ) un estado alcanzable.
   Sea $r \in \mathsf{dom}(\Theta)$. Dado que Θ es Causally Consistent,

   $$CC(\Theta(r))$$

   Además sabemos que $\Theta(r) \subseteq E.\mathsf{A}$, dado que todos los efectos de los stores son parte de la sopa.
   Luego,

   $$\Theta(r) \sqsubseteq E$$

   que combinado con

   $$\forall \mathcal{S} \sqsubseteq E.\mathcal{I}(\mathcal{S})$$

   implica que $\mathcal{I}(\Theta(r))$, tomando $\mathcal{S} = \Theta(r)$.

□

A continuación, buscamos caracterizar las ejecuciones donde toda réplica cumple su invariante. Para eso definimos

Definición 5.2.2 (Ejecución $\mathcal{I}$*-valid*)**.** Decimos que una ejecución abstracta E es $\mathcal{I}$*-valid* si y solo si $\forall \mathcal{S} \sqsubseteq E.\mathcal{I}(\mathcal{S})$.

Vinculando con lo anteriormente definido, nos interesa probar que una especificación completa, incluyendo las operaciones y sus niveles de consistencia, es segura.

Definición 5.2.3 (Especificación $\mathcal{I}$-*valid*)**.** Decimos que una especificación es $\mathcal{I}$-*valid* si y solo si para todo estado alcanzable (E, Θ, Σ), E es $\mathcal{I}$-*valid* .

Como imaginamos, analizar todos los estados alcanzables de una especificación es inviable. Por eso debemos diseñar estrategias modulares para averiguar si una especificación cumple un invariante. Concentrémonos saber si la ejecución de cada operación es segura.

Definición 5.2.4. Decimos que una operación $\mathtt{op}$ es $\mathcal{I}$-*safe* en una especificación con $\Psi(\mathtt{op}) = \tau$, si y solo, para todo estado $\mathcal{I}$-*valid* E, cuando $E \rightarrow^{\eta}_{\mathtt{op},\tau} E'$, E' es $\mathcal{I}$-*valid* .

Necesitamos vinclular la correctitud y seguridad las operaciones, con la seguridad de una especificación completa. Para esto tenemos el siguiente teorema.

Teorema 5.2.2. *Si en una especificación, toda operación* $\mathtt{op}$ *es* $\mathcal{I}$-*safe , entonces la especificación es* $\mathcal{I}$-*valid .*

Demostración. Hay dos tipos de reducciones en la semántica. [EffVis], que no cambia la sopa de efectos. Y $E \rightarrow^{\eta}_{\mathtt{op},\tau} E'$, que al ser $\mathcal{I}$-*safe* , terminan en ejecuciones E' $\mathcal{I}$-*valid* . □

5.3. Proof rules

El objetivo de esta sección es el desarrollo de herramientas para demostrar la seguridad de las ejecuciones. Para eso, nos ayudaremos de todas las definiciones de la sección anterior.

El siguiente lema nos garantiza una regla intuitiva y esperable, que podria resumirse de la siguiente manera: Si no cambiamos nada, seguimos en terreno seguro. En detalle dice que si partimos de una ejecución abstracta segura, y ejecutamos una operación, se llega a una ejecución abstracta nueva E'. Y si tomamos un conjunto de efectos $\mathcal{S} \sqsubseteq E'$ pero que no contenga al nuevo efecto recientemente generado, entonces ese conjunto cumple el invariante. Cumplirá el invariante porque es un conjunto que pertenece a la ejecución original, que era segura.

Lema 5.3.1 (Inclusión Invariantes). Si E es $\mathcal{I}$-*valid* , $E \rightarrow^{\eta}_{\mathtt{op},\tau} E'$, $\mathcal{S} \sqsubseteq E'.A$ y $\eta \notin \mathcal{S}$ entonces $\mathcal{I}(\mathcal{S})$.

Demostración. Sabemos que $\mathcal{S} \sqsubseteq E.\mathsf{A}$. Luego $\mathcal{S} \subseteq E.\mathsf{A} = \{\ \eta\} \cup E'.\mathsf{A}$.

Si además $\eta \notin \mathcal{S}$, $\mathcal{S} \subseteq E.\mathsf{A}$. Luego $\mathcal{S} \sqsubseteq E.\mathsf{A}$.

Por otra parte, dado que E es $\mathcal{I}$-*valid* y $\mathcal{S} \sqsubseteq E.\mathsf{A}$, vale $\mathcal{I}(\mathcal{S})$.

□

Ahora si, estamos en condiciones de analizar si una operación es segura. Para esto proveeremos herramientas teóricas. Los siguientes teoremas garantizan el cumplimiento de un invariante bajo ciertas condiciones sobre la ejecución y transmisión de un efecto.

Teorema 5.3.2. *Sea* (Λ, Ψ) *una especificación donde* $\mathtt{op}$ *es una operación tal que* $\Psi(op) = SC$*. Si además :*

$$\forall \mathcal{S}.\ (\mathcal{I}(\mathcal{S}) \wedge \mathtt{op}(\mathcal{S}) \rightsquigarrow \eta \implies \mathcal{I}(\{\eta\} \cup \mathcal{S})$$

entonces $\mathtt{op}$ *es* $\mathcal{I}$-*safe .*

Demostración. Desenmarañemos las definiciones:

Sea `op` una operación que cumple la hipótesis:

$$\forall \mathcal{S}.\ (\mathcal{I}(\mathcal{S}) \wedge op(\mathcal{S}) \rightsquigarrow \eta \implies \mathcal{I}(\{\eta\} \cup \mathcal{S})$$

y

$$\Psi(op) = SC$$

Sea $E \rightarrow^{\eta}_{\mathtt{op},\tau} E'$. Supongamos que E es $\mathcal{I}$*-valid* , veamos que E' también lo es. Así probaremos que `op` es $\mathcal{I}$*-safe* .

Al ser E $\mathcal{I}$*-valid* sabemos que $\mathcal{I}(\Theta(r))$

Si vale $E \rightarrow^{\eta}_{\mathtt{op},\tau} E'$, como además *op* es de tipo SC, entonces en la réplica donde se generó η estaban presentes todos los efectos.

En la semántica vemos como requerimientos que:

$$E.\mathsf{A} \subseteq \Theta(r)$$

$$op(\Theta(r)) \rightsquigarrow \eta$$

Luego sabemos que $op(A) \rightsquigarrow \eta$ y $\mathtt{vis}^{-1}(\eta) = A$.

Por hipótesis, reemplazando $\mathcal{S}$ por A tenemos que $\mathcal{I}(\{\eta\} \cup A)$.

Sea $\mathcal{S} \sqsubseteq E'$. Consideramos dos casos, uno donde η pertenece a $\mathcal{S}$, y otro donde no.

Si $\eta \in \mathcal{S}$, como $\mathcal{S} \sqsubseteq E'$ entonces $\mathtt{vis}^{-1}(\eta) = A$. Luego $\mathcal{S} = \{\eta\} \cup A$ y sabíamos que $\mathcal{I}(\{\eta\} \cup A)$.

Si $\eta \notin \mathcal{S}$, el teorema 5.3.1 garantiza $\mathcal{I}(\mathcal{S})$.

Entonces E' es $\mathcal{I}$*-valid* . □

Teorema 5.3.3. *Sea* (Λ, Ψ) *una especificación donde* `op` *es una operación tal que* $\Psi(op) = EC$*. Si además :*

$$\forall \mathcal{S}.\ (\mathcal{I}(\mathcal{S}) \wedge \mathtt{op}(\mathcal{S}) \rightsquigarrow \eta) \implies (\forall \mathcal{S}'.\mathcal{S} \subseteq \mathcal{S}' \wedge \mathcal{I}(\mathcal{S}') \implies \mathcal{I}(\{\eta\} \cup \mathcal{S}'))$$

entonces `op` *es* $\mathcal{I}$*-safe ..*

Demostración. Sea $E \rightarrow^{\eta}_{\mathtt{op},\tau} E'$. Supongamos que E es $\mathcal{I}$*-valid* , veamos que E' también lo es. Así probaremos que (op, EC) es $\mathcal{I}$*-safe* .

Al ser E $\mathcal{I}$*-valid* sabemos que $\mathcal{I}(\Theta(r))$.

Sea $\mathcal{S} \sqsubseteq E'.A$.

Si $\eta \notin \mathcal{S}$ entonces por el teorema 5.3.1 $\mathcal{I}(\mathcal{S})$.

Si $\eta \in \mathcal{S}$, sabemos que $\mathcal{S} - \{\eta\} \subseteq E.A$. Dado que $\mathcal{S} \sqsubseteq E'.A$, el conjunto $\mathcal{S}$ conserva `vis`.

$$E'.\mathtt{vis}^{-1}(\eta) \subseteq \mathcal{S}$$

Concluimos entonces que todo lo almacenado en el store, en la réplica r, pertenece a $\mathcal{S}$.

$$E'.\texttt{vis}^{-1}(\eta) = \Theta(r) \subseteq \mathcal{S}$$

y como η es un efecto nuevo,

$$\Theta(r) \subseteq \mathcal{S} - \{\eta\}$$

Además como E es $\mathcal{I}$-*valid* $\mathcal{I}(\mathcal{S} - \{\eta\})$.

En la hipótesis del teorema, reemplazando $\mathcal{S}$ por $\Theta(r)$, y $\mathcal{S}'$ por $\mathcal{S} - \{\eta\}$, obtenemos que vale $\mathcal{I}(\mathcal{S})$

$$(\mathcal{I}(\Theta(r)) \wedge op(\Theta(r)) \rightsquigarrow \eta)) \implies (\Theta(r) \subseteq (\mathcal{S} - \{\eta\}) \wedge \mathcal{I}(\mathcal{S} - \{\eta\}) \implies \mathcal{I}(\mathcal{S}))$$

□

6. PROTOTIPO DE VERIFICACIÓN

Este capítulo se propone implementar una prueba de concepto de las herramientas teóricas vistas anteriormente. Para esto, modelaremos el programa de administración de torneos (ver sección 2.2) en el lenguaje de programación F* (pronunciado F-star) [SHK+16].

Empezaremos por describir la sintaxis y el funcionamiento de este lenguaje en la sección 6.1. Luego en la sección 6.2 vamos a mostrar como modelar el torneo utilizando F* . Más adelante (sección 6.3) expresaremos el invariante en F* y probaremos que el programa lo cumple teniendo en cuenta las *proof rules* (definidas en 5.3). Por último, en 6.4 demostramos que los observadores conmutan y por ende el modelo es válido para su uso en QUELEA.

6.1. Sintaxis de F*

F* es un lenguaje funcional que por su sintaxis pertenece a la familia de lenguajes ML. Además tiene un sistema de tipos muy expresivo, con tipos dependientes que permiten expresar especificaciones funcionales. El *type-checker* verifica que un programa cumple con su especificación descargando las pruebas sobre un SMT solver (usualmente Z3).

Por ejemplo el siguiente programa calcula la cantidad de veces que un elemento de tipo `a` aparece en una lista. Las llaves indican una fórmula lógica que el tipo debe cumplir. $x >= 0$ en este caso indica que el resultado es un natural.

```
val count_of: #a:eqtype -> a -> list a -> r:int {r >= 0}
let rec count_of #a x = function
  | [] -> 0
  | y :: ys -> (if x = y then 1 else 0 ) + count_of x ys
```

El tipo *Lemma* indica una fórmula que el programa debe poder concluir cuando descargue las pruebas sobre el SMT solver. El siguiente programa nos da una demostración de que contar las apariciones de un elemento en una lista de un solo elemento, es uno o cero según el caso.

Es decir, cualquiera sean `x` e `y`, `count_of x [y]` (cantidad de apariciones de `x` en la lista donde solo está `y`) es uno si `x` e `y` eran iguales, y es cero si `x` e `y` eran distintos.

En F* podemos expresarlo de esta forma:

```
val count_of_singleton_2: #a:eqtype -> Lemma (forall x y. (x = y
   ==> count_of #a x [y] = 1) /\ (~(x = y) ==> count_of #a x [y] =
   0) )
let count_of_singleton_2 #a = ()
```

Donde esperamos un resultado de tipo *Lemma*, podemos insertar () (una expresión de tipo *unit*) directamente. F* llamará al SMT solver para tratar de probar el lema que especificamos.

6.2. Ejemplo en F*

En esta sección revisitaremos el ejemplo del torneo modelándolo en F* . Dado que la implementación utiliza tipos algebraicos, la notación será similiar a las utilizada en la sección 2.2.

Recordemos que en este programa tenemos jugadores que pueden inscribirse y desuscribirse de un torneo. Los jugadores se representan mediante un identificador, en este caso un *string*. Además debemos cumplir una condición: En todo momento la cantidad de jugadores inscriptos no debe superar la capacidad

$$\forall\, l \sqsubseteq E.\ \ currentAmountOfPlayers\ \ l \leq capacity$$

Esta condición será el invariante que debemos respetar.

Para poder respetar el invariante, tomamos la decisión de asignarle a la operación de inscripción (`enroll`) consistencia fuerte. De esta forma, solo se agregaran jugadores al torneo si tenemos la certeza que la cantidad actual está por debajo de la capacidad del torneo. La consistencia fuerte garantiza que la operación conoce la cantidad real de inscriptos actuales en todas las réplicas.

Al mismo tiempo, la operación de desuscripción no tiene restricciones tan fuertes. Al propagar el efecto generado a cualquier réplica, sin importar su estado actual, el invariante seguirá siendo válido. Por lo tanto podemos asignarle una consistencia más débil a la operación. En particular utilizamos *causal visibility*, por ser la más débil entre las disponibles en la semántica de QUELEA.

Utilizando el modelo planteando a lo largo de esta sección describiremos la implementación y verificación del programa del modelo. Para empezar desarrollamos el tipo algebraico `tournament_effect`, que se corresponde con un efecto en la teoría que planteamos.

```
let player = string

type tournament_effect =
  | Enroll : player -> tournament_effect
  | Disenroll : player -> tournament_effect
  | Id : tournament_effect
```

Recordamos que `Enroll` es un efecto parametrizado por un jugador, que representa el agregado de un jugador al torneo. El jugador está representado por un identificador, que en este modelo es simplemente un `string`.

A continuación, escribimos los observadores, es decir aquellas funciones que toman listas de efectos y retornan un valor, conformando el estado abstracto del tipo de datos.

La función `was_enrolled` retorna el conjunto de participantes del torneo que fueron inscriptos alguna vez.

```
val was_enrolled : list tournament_effect -> set player
let rec was_enrolled = function
  | [] -> empty
  | (Enroll x :: ts) -> union (singleton x) (was_enrolled ts)
  | (Disenroll x :: ts) -> was_enrolled ts
```

```
  | (Id  :: ts) -> was_enrolled ts

val was_disenrolled : list tournament_effect -> set player
let rec was_disenrolled = function
  | [] -> empty
  | (Enroll x :: ts) -> was_disenrolled ts
  | (Disenroll x :: ts) -> union (singleton x) (was_disenrolled ts)
  | (Id  :: ts) -> was_disenrolled ts
```

`was_enrolled` es definida usando *pattern matching*. Cuando la lista es vacía retornamos el conjunto vacío. En cambio, si la lista se constituye mediante el constructor : que agrega un elemento, distinguimos casos según el efecto agregado.Si se agregó un efecto `enroll`, unimos el jugador correspondiente al resultado.

`was_disenroll` retorna el conjunto de participantes del torneo que fueron desuscriptos alguna vez. Su definicion es analoga a la de `was_enrolled` teniendo en cuenta que agrega a aquellos que fueron desuscriptos en lugar de aquellos que se inscribieron.

Considere las siguientes definiciones para las operaciones `enroll` y `disenroll`.

```
val enroll : l:list tournament_effect {sz l <= capacity} -> player
   -> o: option tournament_effect
let enroll l x =
  if (sz l <= capacity - 1) then Some (Enroll x)
  else None


val disenroll : l:list tournament_effect -> player ->
  o:option tournament_effect
  let disenroll l x = Some (Disenroll x)
```

Contamos con la función `sz`, que calcula la cantidad de jugadores inscriptos en el torneo dado un estado del sistema. Para ver el detalle de su implementación ir al anexo, sección 8.2.

Usando esta función, especificamos las operaciones. La operación enroll toma un estado, y si la cantidad actual de inscriptos al torneo es menor que la capacidad, agrega un jugador. La operación disenroll remueve un jugador del torneo.

6.3. Invariantes

En lo que sigue, damos testigos de que ciertas propiedades valen. Es decir, construcciones en el lenguaje que codifican la validez de una propiedad deseada.

A continuación tenemos la función `enroll_sc`, que prueba que la operación `enroll` es segura bajo la consistencia [SC]. Para probarlo, basta probar que al agregar el efecto generado por la operación tendremos un estado donde la cantidad de jugadores es menor o igual a la capacidad. Debido a las restricciones de la semántica [SC], agregamos el efecto a la misma lista donde se ejecutó la operación.

```
val enroll_sc : l:list tournament_effect {sz l <= capacity}
              -> x:player
              -> Lemma (isSome (enroll l x )
                   ==> sz (get (enroll l x ) :: l) <= capacity)
let enroll_sc l x = ()
```

De esta forma modelamos que la réplica donde se ejecuta `enroll`, debe contener todos los efectos existentes en el sistema. Además modela la *proof-rule* abstracta provista por el teorema 5.3.2.

Ahora, debemos especificar un concepto que hasta ahora hemos eludido. Teniendo en cuenta la semántica de QUELEAestudiada en secciones anteriores, sabemos que la transmisión de efectos cumple **causal visibility**. Es decir, que un efecto está presente en una réplica solo si sus predecesores están presentes. Para poder modelar este fenómeno, describiremos un tipo dependiente, `causal l1 l2`.

El tipo `causal l1 l2` codifica que todos los efectos de l1 están presentes en l2. De esta manera, si un efecto se genera en una réplica con lista de efectos l1, puede ser transmitiada a una réplica con lista de efectos l2.

Para poder escribir la especificación del tipo `causal l1 l2` debemos darnos cuenta que en el modelo que estamos describiendo, no distinguimos entre efectos del mismo tipo generados en momentos distintos. Es decir, dos efectos `Enroll "jugador1"` (un depósito del valor 10) son indistinguibles, sin importar en que contexto fueron generadas.

Teniendo esto en cuenta, no alcanzará con pedir que un efecto que está en l1 también esté en l2. Si l1 contiene dos efectos `Enroll "jugador1"`, también vamos a querer que l2 tenga dos efectos de este tipo. Por esto escribimos la función auxiliar `count\_of`

```
type causal (l1:list tournament_effect) (l2:list
   tournament_effect) =
  (forall y. count_of y l1 <= count_of y l2)
```

En detalle, el tipo `causal l1 l2` especifica lógicamente que para cualquier efecto (`y`), la cantidad de veces que aparece en `l1` (es decir `count_of y l1`) es menor o igual que la cantidad de veces que aparece en `l2` (`count_of y l2`).

Así verificamos que la operación `disenroll` es segura. Teniendo en cuenta el teorema 5.3.3, para que la operación cumpla el invariante, debemos probar que dado un estado `l` de una réplica, agregando el nuevo efecto a cualquier réplica `l'` (que cumpla `causal l l'`) se llega un estado que cumple el invariante.

```
val disenroll_cv : l:list tournament_effect {sz l <= capacity} ->
  x:player -> l':list tournament_effect {causal l l' /\ sz l' <=
   capacity} ->
  Lemma (isSome (disenroll l x) ==> sz (get (disenroll l x) :: l')
   <= capacity )
let disenroll_cv l x l' = ()
```

Esto es contrastante con la situación de enroll, ya que aquí tenemos que tener en cuenta la propagación del efecto generado a réplicas que tengan otros efectos.

6.4. Conmutatividad

Por último, vamos a verificar otra propiedad de nuestro programa: La conmutatividad de los observadores. Para esto verificaremos que dados dos efectos (`tournament_effect`) arbitrarios, agregarlos a una lista en cualquier orden da el mismo resultado.

```
val was_enrolled_comm_lemma : x:tournament_effect
    -> y:tournament_effect
    -> l1:list tournament_effect
    -> Lemma (same_set (was_enrolled (x :: y :: l1)
                       (was_enrolled (y :: x :: l1)))

let was_enrolled_comm_lemma x y l1 = ()
```

En la función anterior, x e y representan efectos arbitrarios, y $l1$ es una lista de efectos. Verificamos que `was_disenrolled (y :: x :: l1) = was_disenrolled (x :: y :: l1)` mediante la función `same_set`. De la misma forma podemos probar que otros observadores conmutan. Ver anexo, sección 8.2.

```
val was_disenrolled_comm_lemma : x:tournament_effect
    -> y:tournament_effect
    -> l1:list tournament_effect
    -> Lemma ( same_set (was_disenrolled (x :: y :: l1))
                        (was_disenrolled (y :: x :: l1)))

let was_disenrolled_comm_lemma x y l1 = ()

val all_enrolled_comm_lemma : x:tournament_effect ->
   y:tournament_effect -> l1:list tournament_effect ->
Lemma (same_set (all_enrolled (x :: y :: l1))
                (all_enrolled (y :: x :: l1)))

let all_enrolled_comm_lemma x y l1 = ()
```

7. CONCLUSIONES

Hemos visto que el razonamiento sobre programas con consistencia débil es difícil. Por ende hay una necesidad de desarrollar herramientas teóricas y prácticas de modelado y verificación de sistemas geo-replicados.La correctitud de los mismos dependen de las garantías ofrecidas por los niveles de consistencia. Por eso, en la literatura hay un creciente interés en desarrollar y extender herramientas sobre lenguajes de programación que nos permitan escribir programas con modelos de consistencia débil. En particular, QUELEA propone un framework operacional que permite declarar y verificar las propiedades de consistencia que los programas deben conseguir.

Vimos como, al extender este framework con *proof-rules* somos capaces de probar el cumplimiento de invariantes en sistemas geo-replicados. Como agregado, las *proof-rules* pueden implementarse en F* , acercándonos a la automatización en el proceso de verificación. De esta forma los programadores pueden asegurar que sus programas son seguros mientras que disfrutan de los beneficios en disponibilidad de consistencias débiles.

7.1. Trabajos relacionados

Gotsman et al. [GYF$^+$16] describen un modelo parecido, donde permiten ejecutar operaciones causally consistent sobre CRDTs u operaciones que sincronizan bajo un sistema de *tokens*. En este modelo, dos operaciones que adquieren el mismo token no pueden ser concurrentes. Es decir que son análogos a un sistema de exclusión mutua. En el mismo trabajo proponen *proof-rules* que permiten probar la satisfacción de invariantes, descargando las pruebas sobre un SMT solver. Sin embargo, existen muchos modelos y conjuntamente muchas definiciones de las propiedades básicas, como eventual consistency.

Bouajjani et al. [BEH14] estudian el problema de verificación en sistemas geo-replicados reduciéndolo al problema de Model Checking. Axiomatizan las propiedades de consistencia en términos de comportamiento de las trazas, y reducen el chequeo de estas propiedades a problemas sobre autómatas.

Nair et al. [NPS19] estudian métodos para probar el cumplimiento de invariantes en State based CRDTs.

Por último, Gomes et al. dan ejemplos de como verificar formalmente un CRDT [GKMB17] y proveen una definición de *Strong Eventual Consistency* en el lenguaje Isabelle.

7.2. Trabajo a futuro

Creemos que es posible explorar a futuro la utilización de *theorem provers* para escribir sistemas geo-replicados y probar propiedades. Más aún, aquellos theorem provers como F* que descargan pruebas sobre SMT solvers pueden presentarnos oportunidades para mejorar el proceso de verificación.

Esta es la línea de pensamiento de Lindsey Kuper y Peter Alvaro [KA19], que argumentan la necesidad de utilizar lenguajes *domain-specific* junto a SMT solvers para verificar propiedades de consistencia débil. Mediante la interacción entre el lenguaje y el SMT solver, podemos diseñar sistemas distribuidos que sean correctos por construcción.

Vimos como utilizar F* para verificar la correctitud de operaciones en un sistema geo-replicado. Dado este caso de éxito, queda pendiente explorar más exhaustivamente la utilización de este tipo de lenguajes para el diseño de herramientas de verificación de programas geo-replicados.

Bibliografía

[BEH14] Ahmed Bouajjani, Constantin Enea, and Jad Hamza. Verifying eventual consistency of optimistic replication systems. volume 49, pages 285–296, 01 2014.

[BG13] Peter Bailis and Ali Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Queue*, 11, 03 2013.

[BGYZ14] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: Specification, verification, optimality. volume 49, pages 271–284, 01 2014.

[BND+14] Valter Balegas, Mahsa Najafzadeh, Sérgio Duarte, Carla Ferreira, Marc Shapiro, Rodrigo Rodrigues, and Nuno Preguiça. Putting the consistency back into eventual consistency. *Proceedings of the 10th European Conference on Computer Systems, EuroSys 2015*, 10 2014.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. volume 41, pages 205–220, 10 2007.

[FZWK17] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. pages 328–343, 04 2017.

[GKMB17] Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. Verifying strong eventual consistency in distributed systems. *Proc. ACM Program. Lang.*, 1(OOPSLA):109:1–109:28, October 2017.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent available partition-tolerant web services. *ACM SIGACT News*, 33, 11 2002.

[GYF+16] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. volume 51, pages 371–384, 04 2016.

[KA19] Lindsey Kuper and Peter Alvaro. Toward domain-specific solvers for distributed consistency. In *SNAPL*, 2019.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, 07 1978.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra — a decentralized structured storage system. *Operating Systems Review*, 44:35–40, 04 2010.

[Mic19] Microsoft. Consistency levels in azure cosmos db, 2019. [Online. Accedido el 17-Nov-2019.].

[NPS19] Sreeja Nair, Gustavo Petri, and Marc Shapiro. Invariant safety for distributed applications. 03 2019.

[SHK+16] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, January 2016.

[SKJ15] K.C. Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. *ACM SIGPLAN Notices*, 50:413–424, 06 2015.

[SPBZ11] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. 01 2011.

[Ter13] Douglas Terry. Replicated data consistency explained through baseball. *Communications of the ACM*, 56, 12 2013.

[VV15] Paolo Vi and Marko Vukolić. Consistency in non-transactional distributed storage systems. *ACM Computing Surveys*, 49, 12 2015.

8. ANEXO

8.1. Ejemplo completo en Haskell

```
import Data.Set
capacity = 3

type Player = String
type Operation a v = [TournamentEff] -> a -> (v, Maybe
   TournamentEff)

data TournamentEff =
    Enroll Player |
    Disenroll Player |
    GetEnrolledCount |
    IsEnrolled deriving (Show, Eq)

enroll :: Operation Player ()
enroll hist p = if (currentAmountOfPlayers hist < capacity )
                then ((), Just $ Enroll p)
                else ((), Nothing)

disenroll :: Operation Player ()
disenroll hist p = ((), Just $ Disenroll p)

getEnrolledCount :: Operation () Int
getEnrolledCount hist _ = (currentAmountOfPlayers hist, Just
   GetEnrolledCount)

isEnrolled :: Operation Player Bool
isEnrolled hist p = (isEnrolledObs hist p, Just IsEnrolled)

wasDisenrolled :: [TournamentEff] -> Set Player
wasDisenrolled hist = fromList [ p2 | Disenroll p2 <- hist]

wasEnrolled :: [TournamentEff] -> Set Player
wasEnrolled hist = fromList [ p2 | Enroll p2 <- hist]

allEnrolled :: [TournamentEff] -> Set Player
allEnrolled hist = (wasEnrolled hist \\ wasDisenrolled hist)

isEnrolledObs :: [TournamentEff] -> Player -> Bool
isEnrolledObs hist p = elem p (allEnrolled hist)
```

```
currentAmountOfPlayers :: [TournamentEff] -> Int
currentAmountOfPlayers hist = size (allEnrolled hist)
```

8.2. Código: Ejemplo completo en F*

```
module Tournament
open FStar.Set
open FStar.Option

let player = string

type tournament_effect =
  | Enroll : player -> tournament_effect
  | Disenroll : player -> tournament_effect
  | Id : tournament_effect

val was_enrolled : list tournament_effect -> set player
let rec was_enrolled = function
  | [] -> empty
  | (Enroll x :: ts) -> union (singleton x) (was_enrolled ts)
  | (Disenroll x :: ts) -> was_enrolled ts
  | (Id  :: ts) -> was_enrolled ts

val was_disenrolled : list tournament_effect -> set player
let rec was_disenrolled = function
  | [] -> empty
  | (Enroll x :: ts) -> was_disenrolled ts
  | (Disenroll x :: ts) -> union (singleton x) (was_disenrolled ts)
  | (Id  :: ts) -> was_disenrolled ts

val difference : #a:eqtype -> set a -> set a -> set a
let difference #a s1 s2 = intersect s1 (complement #a s2)

val all_enrolled : list tournament_effect -> set player
let all_enrolled ts = difference (was_enrolled ts)
   (was_disenrolled ts)

val remove : #a:eqtype -> a -> set a -> set a
let remove #a x s = intersect s (complement (singleton x ))


val get : o:option 'a {isSome o} -> 'a
let get (Some x) = x

type same_set (l1:set player) (l2: set player) = (forall q. mem q
   l1 = mem q l2)
```

```
val current_state : l:list tournament_effect -> r:(set player *
    set player * int )
let rec current_state = function
    | [] -> (empty , empty , 0)
    | (Enroll x :: ls ) ->
      let (e , r , sz) = current_state ls in
      if (mem x e) then (e , r , sz)
      else if (mem x r) then (e , r , sz)
      else (union (singleton x) e, r, sz + 1)
    | (Disenroll x :: ls ) ->
      let (e , r , sz) = current_state ls in
      if (mem x r) then (e , r , sz)
      else if (mem x e) then (remove x e , union (singleton x) r ,
    sz - 1)
      else (e, union (singleton x) r, sz)
    | (Id :: ls ) -> current_state ls


// Esto es equivalente currentAmountOfPlayers
val sz : list tournament_effect -> int
let sz l = let (_, _ , s ) = current_state l in s

val capacity : int
let capacity = 3

val enroll : l:list tournament_effect {sz l <= capacity} -> player
    -> o: option tournament_effect
let enroll l x =
  if (sz l <= capacity - 1) then Some (Enroll x)
  else None

val enroll_sc : l:list tournament_effect {sz l <= capacity} ->
    x:player -> Lemma (isSome (enroll l x ) ==> sz (get (enroll l x
    ) :: l) <= capacity)
let enroll_sc l x = ()

val count_of: #a:eqtype -> a -> list a -> nat
let rec count_of #a x = function
  | [] -> 0
  | y :: ys -> (if x = y then 1 else 0 ) + count_of x ys

type permutation (#a:eqtype)(l2:list a) = l1:list a {forall y.
    count_of y l1 = count_of y l2}

type causal (l1:list tournament_effect) (l2:list
    tournament_effect) =
  (forall y. count_of y l1 <= count_of y l2)

val disenroll : l:list tournament_effect -> player ->
```

```
  o:option tournament_effect
let disenroll l x = Some (Disenroll x)

val disenroll_cv : l:list tournament_effect {sz l <= capacity} ->
  x:player -> l':list tournament_effect {causal l l' /\ sz l' <=
   capacity} ->
  Lemma (isSome (disenroll l x) ==> sz (get (disenroll l x) :: l')
   <= capacity )
let disenroll_cv l x l' = ()

val was_enrolled_comm_lemma : x:tournament_effect ->
   y:tournament_effect -> l1:list tournament_effect -> Lemma
   (forall q. (mem q (was_enrolled (x :: y :: l1)) = mem q
   (was_enrolled (y :: x :: l1))))
let was_enrolled_comm_lemma x y l1 = ()

val was_disenrolled_comm_lemma : x:tournament_effect ->
   y:tournament_effect -> l1:list tournament_effect -> Lemma
   (forall q. (mem q (was_disenrolled (x :: y :: l1)) = mem q
   (was_disenrolled (y :: x :: l1))))
let was_disenrolled_comm_lemma x y l1 = ()

let first (s1, s2, s3 ) = s1
let second (s1, s2, s3 ) = s2
let third (s1, s2, s3 ) = s3

val current_state_1_comm_lemma : x:tournament_effect ->
   y:tournament_effect -> l1:list tournament_effect -> Lemma (
(same_set (first (current_state (x :: y :: l1)))
          (first (current_state (y :: x :: l1)))))

let current_state_1_comm_lemma x y l1 = ()

val current_state_2_comm_lemma : x:tournament_effect ->
   y:tournament_effect -> l1:list tournament_effect -> Lemma (
(same_set (second (current_state (x :: y :: l1)))
          (second (current_state (y :: x :: l1)))))

let current_state_2_comm_lemma x y l1 = ()

val all_enrolled_comm_lemma : x:tournament_effect ->
   y:tournament_effect -> l1:list tournament_effect -> Lemma (
(same_set (all_enrolled (x :: y :: l1))
          (all_enrolled (y :: x :: l1))))

let all_enrolled_comm_lemma x y l1 = ()
```