Universidad Nacional de Córdoba
Facultad de Matemática, Astronomía y Computación

VERIFICACIÓN DE LÓGICAS MODALES DINÁMICAS EN COQ

FRANCISCO CARLOS TRUCCO

Directores:
DR. RAUL FERVARI
DR. BETA ZILIANI

Trabajo Especial de la Licenciatura en Ciencias de la Computación

Marzo 2019

If I had a world of my own, everything would be nonsense.
Nothing would be what it is, because everything would be what it isn't.
And contrary wise, what is, it wouldn't be.
And what it wouldn't be, it would. You see?

— Lewis Carroll,
Through the Looking-Glass and What Alice Found There

The theorem can be likened to a pearl, and the method of proof to an oyster.
The pearl is prized for its luster and simplicity; the oyster is a complex living
beast whose innards give rise to this mysteriously simple gem.

— Douglas R. Hofstadter,
Gödel, Escher, Bach: An Eternal Golden Braid

RESUMEN

Los lenguajes modales son lenguajes adecuados para describir propiedades de grafos dirigidos con nodos etiquetados. Estas estructuras aparecen en una gran variedad de problemas de diversas áreas del conocimiento. Como consecuencia de esto, existe una gran variedad de lógicas modales. En la práctica cada vez que se define una nueva lógica modal, muchos teoremas deben ser demostrados y revisados nuevamente para esta nueva lógica. Este proceso, además de resultar tedioso, es propenso a errores. Afortunadamente, la tarea de realizar demostraciones complejas ha comenzado cada vez más a ser asistida por herramientas computacionales. En particular, el asistente de prueba Coq ha sido utilizado para demostrar problemas que permanecieron irresolutos durante muchos años. En este trabajo, formalizamos y verificamos en el asistente de demostraciones Coq un teorema de invarianza bajo bisimulación de ciertas lógicas modales con operadores dinámicos capaces de modificar la relación de accesibilidad. Esto constituye un primer paso en la aplicación de un demostrador interactivo para demostrar propiedades más complejas para diferentes lógicas modales.

ABSTRACT

Modal languages are adequate languages to describe properties of labeled directed graphs. These structures appear in a wide variety of problems from different areas of knowledge. As a consequence, there exist a great number of modal logics. In practice, every time a new modal logic is defined, many theorems must be proved and peer-reviewed again for this new logic. This process, besides being tedious, is error prone. Fortunately, the task of writing complex proofs has increasingly begun to be assisted by computational tools. In particular, the Coq proof assistant has been successfully used to prove theorems that remained unsolved for many years. In this work, we formalize and verify an invariance under bisimulation result in the proof assistant Coq for a particular family of dynamic modal logics that change the accessibility relation of a model. This is a first step towards the application of an interactive proof assistant in the formalization and verification of complex properties of different modal logics.

AGRADECIMIENTOS

ÍNDICE GENERAL

Parte I

UNA INTRODUCCIÓN

1 INTRODUCCIÓN

Las lógicas modales [10] son lenguajes adecuados para describir propiedades de estructuras relacionales también conocidas como grafos. El lenguaje modal básico es una extensión conservativa de la lógica proposicional con el operador $\Diamond$, que permite, a partir de un punto de evaluación en el modelo relacional, describir propiedades de los estados accesibles desde el punto de evaluación.

Una particularidad de los lenguajes modales es su gran variedad, ya que en general es posible definir un nuevo operador, darle la semántica y estudiarlo formalmente, de acuerdo a las necesidades particulares. Dado que estos lenguajes en general pueden codificarse en algún lenguaje clásico (tal como la lógica de primer orden), es posible estudiar a los lenguajes modales como fragmentos de lógica clásica. Una propiedad interesante para investigar, es el poder expresivo de los mismos, ya que nos permite encontrar conexiones con fragmentos de otros lenguajes más conocidos. Una de las herramientas clásicas para investigar el poder expresivo de los lenguajes es la bisimulación. Una bisimulación es una relación entre modelos, con ciertas condiciones para garantizar la expresividad de cierta lógica. El teorema fundamental que se busca garantizar con las bisimulaciones, conocido como Teorema de Invarianza por Bisimulación, establece que si dos modelos son bisimilares (para cierta lógica L) entonces tales modelos satisfacen las mismas fórmulas de L.

Recientemente, la tarea de realizar demostraciones complejas ha comenzado cada vez más a ser asistida por herramientas computacionales. En particular, el asistente de prueba Coq ha sido utilizado para demostrar problemas que permanecieron irresolutos durante muchos años. En este trabajo formalizamos y verificamos en el asistente de demostraciones Coq un teorema de invarianza bajo bisimulación de ciertas lógicas modales dinámicas desarrolladas en [3]. Para lograr esto, adaptamos la formalización del lenguaje modal básico introducida en [33] para luego formalizar la noción de bisimulación para la lógica modal básica y demostrar usando dicho asistente de prueba, su correspondiente teorema de invarianza. Luego adaptamos esta formalización a las lógicas dinámicas descriptas en [3] verificando el teorema de invarianza bajo bisimulación para estas lógicas. Este trabajo constituye un primer paso en la aplicación de un demostrador interactivo para demostrar propiedades más complejas para diferentes lógicas modales.

1.1 VERIFICACIÓN DE DEMOSTRACIONES

Formalmente, una demostración correcta es simplemente una sucesión de aplicaciones válidas de pasos o de reglas de inferencia. Una regla de inferencia es simplemente una manipulación simbólica sobre enunciados que preserva la

verdad de los mismos. Esto quiere decir que si aplicamos una regla de inferencia sobre un enunciado verdadero, el nuevo enunciado obtenido también es verdadero. De esta forma, si partimos de enunciados verdaderos y sólo aplicamos reglas de inferencias válidas sólo llegaremos a nuevos enunciados verdaderos.

Sin embargo, en las demostraciones matemáticas por lo general no se explicitan todas las aplicaciones de las reglas de inferencia. En cambio, los matemáticos expresan informalmente cuál es la estructura de la prueba. Por este motivo, las verificaciones de los teoremas son informales. De hecho, estas verificaciones son realizadas mediante la revisión por pares. Sin embargo, este método de verificación presenta algunos inconvenientes. Por ejemplo, si las demostraciones son muy extensas es muy probable que un error no sea detectado en el proceso y por lo tanto la demostración no se considerará completamente convincente dentro de la comunidad matemática. Si pudieramos formalizar las demostraciones y verificar que cada una de las aplicaciones de las reglas de inferencia es correcta, tendríamos una mayor certeza de la correctitud de los resultados matemáticos. Sin embargo, esto no resuelve el problema de la falibilidad humana, de hecho sólo lo complica puesto que las demostraciones formales son por lo general mucho más largas y menos comprensibles que su contraparte informal.

Afortunadamente, las computadoras son una herramienta muy adecuada para verificar si una sucesión de aplicaciones de reglas de inferencia son válidas. Este hecho ha motivado numerosas investigaciones [7, 8, 16, 17, 26, 29] en las cuales importantes resultados matemáticos fueron formalizados y verificados mediante el uso de programas denominados asistentes de prueba. Un asistente de prueba no sólo consiste en un verificador de demostraciones, si no que también le provee al usuario una serie de funcionalidades que facilitan el desarrollo de demostraciones, como por ejemplo la capacidad de automatizar partes de las pruebas y de construirlas interactivamente. Algunos de los asistentes de pruebas más conocidas son Coq [9], Agda [12], Isabelle [28], Lean [27] y HOL4 [32].

Una de las aplicaciones más interesantes de estos programas ha sido la verificación formal de especificaciones de software. Dado que las demostraciones que prueban que un software satisface una cierta especificación son demostraciones matemáticas, es evidente que estos programas también servirán para estos propósitos. De hecho, la literatura en el tema es muy extensa. Entre las aplicaciones de estos asistentes de demostraciones encontramos la verificación de compiladores [23], de analizadores estáticos de código [20], de protocolos de criptográficos [25] y de sistemas operativos [22].

1.2 COQ

Coq es un asistente de demostraciones interactivo. Este asistente de demostraciones posee un lenguaje formal que le permite al usuario formalizar definiciones matemáticas, algoritmos, teoremas y sus correspondientes demostraciones. Por otra parte, el asistente de demostraciones permite construir las demostraciones de manera interactiva.

La lógica subyacente que permite especificar las demostraciones de los teoremas es una lógica intuicionista con tipos dependientes conocida como Cálculo

de Construcciones Inductivas [9]. Gracias a la correspondencia de Curry Howard [19] podemos interpretar las proposiciones de esta lógica como tipos y las demostraciones de estas proposiciones como programas que tienen el tipo de las proposición que demuestran. Esto tiene una consecuencia importante para la verificación de las demostraciones puesto que demostrar un teorema es equivalente a verificar que el tipo de la demostración (o programa) es efectivamente la proposición que se quiere demostrar. Por este motivo el verificador de teoremas de Coq es esencialmente un verificador de tipos.

1.3 LÓGICAS MODALES DINÁMICAS

La lógica modal tiene sus orígenes en los trabajos de C. I. Lewis, un filósofo y lógico estadounidense que en 1918 intentó formalizar la noción intuitiva del operador de implicación mediante los conceptos de necesidad y posibilidad [24]. Estas lógicas modales pronto se convirtieron en objeto de interés para investigadores de diversas áreas como computación, filosofía, matemática, economía, inteligencia artificial, teoría de juegos, entre otras.

Las lógicas modales son lenguajes interesantes porque permiten expresar propiedades de grafos dirigidos etiquetados, estructuras que aparecen en una gran variedad de problemas de diversas áreas del conocimiento. Es por este motivo, que en la literatura existen una gran variedad de lógicas modales, con otros modos de verdad diferentes a los de necesidad y posibilidad. Las lógicas epistémicas [18, 34] por ejemplo, nos permiten formalizar los conceptos de conocimiento y creencia. Las lógicas temporales [30] nos permiten hablar de si un evento ocurre en algún punto en el futuro y de si un evento ocurre en el futuro siempre desde un punto en adelante. Sin embargo, todas estas lógicas sólo permiten explorar ciertas propiedades de la estructura de un grafo pero no modificarla. Por otro lado, existen situaciones en las que la estructura del grafo cambia. En [1, 15] los autores desarrollan ciertas lógicas modales, que en el resto del trabajo denominaremos lógicas modales dinámicas, que nos permiten modelar estas situaciones en donde modificaciones en la estructura son requeridas. Los autores resuelven este problema capturando el comportamiento dinámico internamente en el lenguaje. Es decir, definen ciertos operadores modales dinámicos que son capaces de modificar la estructura del grafo. En particular, definen operadores dinámicos capaces de modificar la relación de accesibilidad (esto es, las flechas del grafo dirigido).

1.4 ESTRUCTURA DEL TRABAJO

Este trabajo está divido en dos partes. La primera es una introducción a las lógicas modales básicas y dinámicas y al asistente de demostraciones Coq. La segunda parte es una descripción detallada de la formalización de las lógicas modales básica y dinámica en Coq. En los apéndices el lector puede encontrar todo el código fuente del trabajo.

En el Capítulo 2 definimos la sintaxis y la semántica de lógica modal básica, en conjunto con algunas definiciones como el concepto de bisimulación y de equivalencia modal que serán necesarias para demostrar el teorema de invarianza bajo bisimulación. En el Capítulo 3 definimos de forma análoga todos los conceptos desarrollados en el Capítulo 2 para las lógicas modales dinámicas. En el Capítulo 4 introducimos las herramientas de Coq que fueron utilizadas para formalizar las lógicas modales presentadas en los Capítulos 2 y 3.

En la segunda parte del trabajo, dividimos la formalización en dos pasos. El primero, presentado en el Capítulo 5, constituye la formalización de la lógica modal básica, mientras que el segundo, presentado en el Capítulo 6, constituye la formalización análoga de la lógica modal dinámica. En ambos capítulos además de formalizar la sintaxis y la semántica de las lógicas, se presenta la demostración verificada del teorema de invarianza bajo bisimulación para cada lógica modal.

2 LÓGICA MODAL BÁSICA

En este capítulo introducimos tanto la sintaxis del lenguaje de la lógica modal básica, como algunas definiciones semánticas que utilizaremos luego para poder enunciar y demostrar el teorema de invariancia bajo bisimulación de la lógica modal básica.

2.1 EL LENGUAJE DE LA LÓGICA MODAL BÁSICA

Definición 2.1. *Sea* PROP *un conjunto infinito numerable de símbolos proposicionales, cuyos elementos denotaremos con* $p_0, p_1, \ldots$. *Definimos el lenguaje modal básico* $\mathcal{ML}$ *como:*

$$\varphi ::= p \mid \bot \mid \varphi \to \varphi \mid \Diamond\varphi,$$

donde $p \in$ PROP.

Esta definición nos dice que toda fórmula o bien es un símbolo proposicional, la constante proposicional falso, una implicación o bien es una fórmula con un diamante prefijo.

Notación 2.1. *Dadas dos fórmulas* $\varphi, \psi \in \mathcal{ML}$ *definimos como abreviaciones las siguientes expresiones:*

$$\begin{aligned}
\neg\varphi &:= \varphi \to \bot \\
\top &:= \neg\bot \\
\varphi \wedge \psi &:= \neg(\varphi \to \neg\psi) \\
\varphi \vee \psi &:= \neg\varphi \to \psi \\
\varphi \leftrightarrow \psi &:= (\varphi \to \psi) \wedge (\psi \to \varphi) \\
\Box\varphi &:= \neg\Diamond\neg\varphi
\end{aligned}$$

A través de estas abreviaciones agregamos los operadores de la lógica proposicional y un nuevo operador $\Box$. De la misma forma que el cuantificador existencial de la lógica de primer orden tiene como dual a el cuantificador universal, este nuevo operador $\Box$ es el dual de $\Diamond$, como puede verse en su definición.

2.2 MODELOS DE KRIPKE

A lo largo de este trabajo estaremos utilizando las lógicas modales desde una perspectiva estrictamente semántica. Es decir que pensaremos las lógicas modales como herramientas para hablar de ciertas estructuras o modelos.

Si bien existen muchas formas de dar semántica a las lógicas modales en función de las estructuras de las que queramos hablar [10], en este trabajo nos enfocamos exclusivamente en utilizar las lógicas modales como herramientas para expresar propiedades sobre grafos. Esto quiere decir que los modelos de nuestro lenguaje serán grafos. En particular serán grafos dirigidos con nodos etiquetados.

Definición 2.2. *Un modelo de Kripke o modelo relacional es una tripla* $\mathcal{M} = \langle W, R, V \rangle$*, donde* W *es un conjunto no vacío,* R *es una relación binaria sobre* W *(esto decir que* $R \subseteq W \times W$*) y* V *es una función* $V : \mathit{PROP} \to 2^W$*. Llamaremos dominio a* W*, estados o puntos a los elementos de* W*, relación de accesibilidad a* R *y función de valuación a* V*.*

Un pointed model *es un par* $(\mathcal{M}, w)$ *donde* $\mathcal{M} = \langle W, R, V \rangle$ *es un modelo de Kripke y* $w \in W$*.*

Ejemplo 2.1. Sean $W = \mathbb{N}$, $R = \{(n, n+1) \mid n \in \mathbb{N}\}$ y $V(p_i) = \{i\}$ con $i \in \mathbb{N}$. $\mathcal{M} = \langle W, R, V \rangle$ es un modelo de Kripke. En este caso la relación de accesibilidad es simplemente la relación de siguiente y la valuación está determinada por el hecho de que en cada mundo $w \in W$, $w \in V(p_w)$ y w no pertenece a ningún $V(p)$ para cualquier p distinto de p_w.

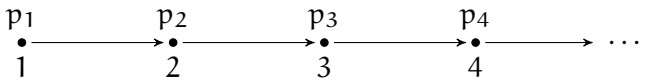

Figura 2.1: El modelo de Kripke $\mathcal{M}$. Cada nodo representa un elemento de W. Dibujamos una flecha entre dos nodos w, v si $(w, v) \in R$. Debajo o sobre cada nodo $w \in W$ escribimos los símbolos proposicionales p tales que $w \in V(p)$.

A veces es útil pensar que cada elemento de W es un estado o un mundo y que las flechas dadas por la relación de accesibilidad R nos permiten movernos de un estado a otro. Por otro lado, dado un estado $w \in W$ podemos pensar a los símbolos proposicionales p tales que $w \in V(p)$ como aquellos que *valen* en ese mundo, o equivalentemente, podemos pensar que $V(p)$ es el conjunto de mundos en donde p es válido.

Ejemplo 2.2. Sean $W = \mathbb{N}$, $R = \{(n, n+1) \mid n \in \mathbb{N}\}$, $w = 0 \in W$ y $V(p) = \varnothing$ para todo $p \in \mathtt{PROP}$. $\mathcal{M} = \langle W, R, V \rangle$ es un modelo de Kripke y $(\mathcal{M}, w)$ es un *pointed model*.

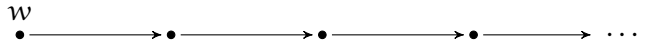

Figura 2.2: El modelo de Kripke $\mathcal{M}$. La mayoría de las veces ignoraremos el nombre de los nodos cuando no sean relevantes. Si la valuación no es de interés tampoco la representaremos al dibujar el modelo, o sólo escribiremos algunos símbolos proposicionales que sean de interés.

2.3 SATISFACIBILIDAD

A continuación introduciremos el concepto de satisfacibilidad modal. Este concepto determinará el significado que le daremos a las fórmulas del lenguaje modal básico. Es decir, es en esta relación de satisfacibilidad que quedará determinada la semántica del lenguaje modal básico.

Definición 2.3. *Sea* $\mathcal{M} = \langle W, R, V \rangle$ *un modelo,* $w \in W$ *un punto y* $\varphi \in \mathcal{ML}$*. Diremos que* $\mathcal{M}, w$ *satisface* φ *y escribiremos* $\mathcal{M}, w \models \varphi$*, si:*

$$\begin{aligned}
\mathcal{M}, w &\models p \text{ sii } w \in V(p) \\
\mathcal{M}, w &\models \bot \text{ nunca} \\
\mathcal{M}, w &\models \varphi \to \psi \text{ sii no se cumple } \mathcal{M}, w \models \varphi \text{ o se cumple } \mathcal{M}, w \models \psi \\
\mathcal{M}, w &\models \Diamond\varphi \text{ sii para algún } v \in W \text{ t.q. } Rwv, \text{ se cumple } \mathcal{M}, v \models \varphi
\end{aligned}$$

Siempre que no ocurra $\mathcal{M}, w \models \varphi$ *escribiremos* $\mathcal{M}, w \not\models \varphi$*.*

Una característica notoria de la noción de satisfactibilidad modal que acabamos de introducir es que nos permite hablar de los modelos desde su interior. En las lógicas de primer orden las proposiciones hablan de los modelos desde el exterior de los mismos. Una fórmula de primer orden o bien es cierta para todo el modelo o bien es falsa para todo el modelo. En las lógicas modales, en cambio, evaluamos las formulas en un punto particular del modelo.

Como dijimos anteriormente, es nuestra intención poder expresar propiedades de los modelos de Kripke utilizando fórmulas del lenguaje modal básico. El lector podría preguntarse en función de esto si las lógicas modales básicas son capaces de expresar cualquier propiedad de los modelos de Kripke y si no lo son, qué propiedades sí son capaces de expresar. Si bien hasta que no introduzcamos el concepto de bisimulación no podremos dar una respuesta más completa a esta pregunta, intentaremos proveerle al lector, mediante ejemplificación, una idea del tipo de propiedades que las lógicas modales básicas pueden expresar.

Ejemplo 2.3. Sea $\mathcal{M} = \langle W, R, V \rangle$ el modelo de Kripke tal que $W = \{w_1, w_2, w_3\}$, $R = \{(w_1, w_2), (w_2, w_3), (w_3, w_1)\}$, $V(p_1) = \{w_1\}$ y $V(p) = \varnothing$ para todo $p \in \texttt{PROP}$ distinto a p_1. Sea $(\mathcal{M}, w_1)$ un *pointed model*.

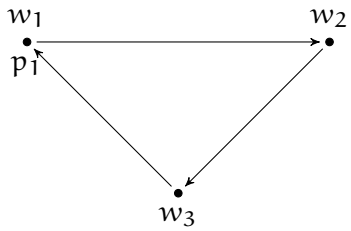

Figura 2.3: El modelo de Kripke $\mathcal{M}$.

Analicemos si las siguientes fórmulas son válidas en este *pointed model*:

▷ $\varphi = p_1$
Puesto que $w_1 \in V(p_1)$ es claro que $\mathcal{M}, w_1 \models \varphi$.

▷ $\varphi = \Diamond p_1$

Basta ver que existe un punto v tal que $(w_1, v) \in R$ y $v \in V(p_1)$. En este caso el único candidato es w_2, por lo que debemos verificar si $w_2 \in V(p_1)$. Pero por la definición de V sabemos que esto es falso. Por lo tanto $\mathcal{M}, w_1 \not\models \varphi$.

▷ $\varphi = \Diamond\Diamond\Diamond p_1$

Formalmente podemos ver que la fórmula es cierta pues w_2, w_3 y w_1 satisfacen $(w_1, w_2) \in R$, $(w_2, w_3) \in R$, $(w_3, w_1) \in R$ y $w_1 \in V(p_1)$. Pero resulta más interesante la interpretación que surje al observar que la fórmula va a ser verdadera si y sólo si existe un nodo a tres saltos en el que vale p_1.

Ejemplo 2.4. ¿Cuál es el significado de la fórmula $\varphi = \Diamond\top$? O más bien ¿qué *pointed models* satisfacen esta fórmula? Dado un *pointed model* $(\mathcal{M}, w)$, notemos que $\mathcal{M}, w \models \varphi$ será verdadero siempre que exista un v en el modelo tal que w y v estén relacionados por la relación de accesibilidad. Por lo que esta fórmula sólo se cumple cuando w tiene al menos un sucesor.

Ejemplo 2.5. ¿Cuál es el significado de la fórmula $\varphi = \Box\psi$? Sea $\mathcal{M} = \langle W, R, V\rangle$ un modelo de Kripke y $w \in W$ un punto. Notemos que por la definición de $\Box$, $\mathcal{M}, w \models \varphi$ será verdadero siempre que para todo v relacionado con w se cumpla que $\mathcal{M}, v \models \psi$.

Podemos pensar la semántica de una fórmula modal como un autómata ubicado dentro de un grafo en un punto determinado, que debe seguir las instrucciones dadas por la fórmula modal. Estas instrucciones le indicarán al autómata que realice ciertos movimientos dentro del grafo. Si el autómata no puede seguir las instrucciones es porque la fórmula no es satisfacible.

A continuación damos un ejemplo que ilustra la interpretación del autómata:

Ejemplo 2.6. Sea $p \in \texttt{PROP}$ y sea $\mathcal{M} = \langle W, R, V\rangle$ el modelo de Kripke de la figura. Sean $\varphi = \Diamond\Diamond\Box p$ y $\psi = \Box\Diamond\Diamond p$. Analicemos el valor de verdad de φ y ψ para el *pointed model* $(\mathcal{M}, w_1)$.

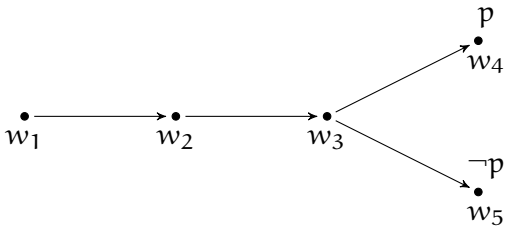

Figura 2.4: El modelo de Kripke $\mathcal{M}$.

Empecemos analizando φ. Imaginemos un autómata ubicado en w_1. La fórmula φ le indica al autómata que debe moverse a un nodo que esté a dos saltos y desde allí debe visitar todos los nodos sucesores y verificar que se cumpla p en esos nodos sucesores. Notemos que esto no es posible pues el único nodo a dos saltos es w_3 y si el autómata se mueve a w_3 deberá visitar todos los sucesores de este nodo y verificar que se satisface p, pero w_5 es un sucesor de w_3 que no satisface p. Ahora continuemos con ψ. Nuevamente, imaginemos a nuestro autómata ubicado

en w_1. La fórmula ψ le indica que debe poder moverse a cualquier sucesor de w_1 y desde allí moverse a algún nodo a dos saltos y verificar p en ese nodo. A diferencia del caso anterior el autómata sí puede lograr ésto. Notar que como w_1 sólo tiene como sucesor a w_2 basta ver que el autómata puede elegir desde w_2 un camino de largo dos tal que desemboque en un nodo en el que valga p. Pero para esto basta con que elija el nodo w_3 y luego el nodo w_4.

2.4 BISIMULACIONES

Los ejemplos de la sección anterior muestran que las lógicas modales pueden expresar propiedades tales como "existe un nodo a 3 saltos tal que satisface tal condición", "todo sucesor satisface cierta condición", "existe algún sucesor que satisface tal condición" o "este nodo no tiene sucesores que satisfagan cierta condición". Pero estos ejemplos proveen una mera descripción informal que no es lo suficiente precisa como para caracterizar el poder expresivo de nuestro lenguaje. Debemos pensar alguna forma de determinar qué puede ser expresado en nuestro lenguaje.

Otro enfoque que podemos tomar es el siguiente. En vez de intentar responder a la pregunta de qué puede ser expresado, intentemos responder a la pregunta de qué *pointed models* puede diferenciar nuestro lenguaje. Esto es, dados dos *pointed models*, tratemos de responder si existe alguna fórmula que sólo sea verdadera en uno de ellos.

Ejemplo 2.7. Sean $\mathcal{M} = \langle W, R, V\rangle$ y $\mathcal{M}' = \langle W', R', V'\rangle$ los modelos de Kripke de la Figura 2.5 con $V(p) = V'(p) = \varnothing$ para todo $p \in \texttt{PROP}$. Los *pointed models* $(\mathcal{M}, w)$ y $(\mathcal{M}', w')$ no pueden ser distinguidos en el lenguaje modal básico puesto que v no es visible desde w. Más precisamente, no existe ninguna fórmula modal que se satisfaga en uno pero no en el otro. Sin embargo, notemos que los *pointed models* $(\mathcal{M}, v)$ y $(\mathcal{M}', w')$ sí pueden ser distinguidos por la fórmula $\varphi = \Diamond\top$ puesto que $\mathcal{M}, v \models \varphi$ y $\mathcal{M}', w' \not\models \varphi$. Podemos concluir dos cosas de esto. Por un lado, que existen modelos que no pueden ser diferenciados por la lógica modal básica y por otro lado, que los puntos de evaluación son importantes a la hora de determinar si dos modelos pueden ser diferenciados o no.

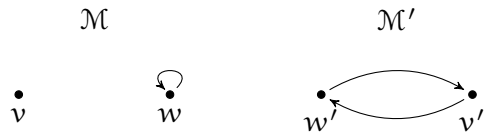

Figura 2.5: Los modelos de Kripke $\mathcal{M}$, $\mathcal{M}'$

Este ejemplo sugiere la existencia de una relación de equivalencia entre *pointed models* que no pueden ser distinguidos por fórmulas modales. Efectivamente, tal equivalencia existe y la denominamos equivalencia modal.

Definición 2.4. *Sean $\mathcal{M} = \langle W, R, V\rangle$ y $\mathcal{M}' = \langle W', R', V'\rangle$ dos modelos, $w \in W$ y $w' \in W'$ dos puntos. Diremos que $\mathcal{M}, w$ y $\mathcal{M}', w'$ satisfacen las mismas fórmulas de*

$\mathcal{ML}$ o que son equivalentes modalmente y escribiremos $\mathcal{M}, w \equiv_{\mathcal{ML}} \mathcal{M}', w'$ si se cumple que para toda $\varphi \in \mathcal{ML}$:

$$\mathcal{M}, w \models \varphi \textit{ sii } \mathcal{M}', w' \models \varphi.$$

Volvamos al problema de caracterizar el poder expresivo de los lenguajes modales. Una pregunta a considerar es cuándo dos modelos poseen ciertas similitudes estructurales que implican su equivalencia modal. Existen muchas relaciones estructurales entre modelos bajo las cuales las fórmulas modales son invariantes, es decir, que hacen a los modelos equivalentes modalmente. Por ejemplo, si dos modelos son *el mismo modelo*, es evidente que deben ser equivalentes modalmente. ¿Pero cuándo dos modelos son *el mismo*? Una de las respuestas posibles es decir que dos modelos el mismo cuando son isomorfos.

Definición 2.5. *Sean $\mathcal{M} = \langle W, R, V\rangle$ y $\mathcal{M}' = \langle W', R', V'\rangle$ dos modelos. Sea $f : \mathcal{M} \to \mathcal{M}'$ una función biyectiva. Diremos que f es un isomorfismo entre $\mathcal{M}$ y $\mathcal{M}'$ cuando:*

- *Para todo $p \in \mathtt{Prop}$ y para todo $w \in W$ se cumple que $w \in V(p)$ si y sólo si $f(w) \in V'(p)$.*
- *Para todo $w, v \in W$ se cumple que $(w, v) \in R$ si y sólo si $(f(w), f(v)) \in R'$.*

Cuando exista un isomorfismo entre $\mathcal{M}$ y $\mathcal{M}'$ diremos que estos modelos son isomorfos.

Como se puede ver en los modelos del Ejemplo 2.7, no es necesario que dos modelos sean isomorfos para que las sentencias que son válidas en uno sean válidas en el otro. Por lo que si bien la relación de isomorfismo nos da una respuesta a la pregunta de cuándo dos modelos son lo suficientemente similares estructuralmente como para que en ambos valgan las mismas sentencias, esta relación no tiene en cuenta ni la expresividad ni la localidad del punto de evaluación que son particulares a las lógicas modales. En este sentido, el concepto de isomorfismo no captura la esencia de los lenguajes modales. Mas aún, es demasiado fuerte.

Bajo estas consideraciones surge naturalmente la pregunta de qué otras relaciones de invarianza estructural entre modelos podemos definir que sean más *modales*. Si bien no hay una única respuesta, el concepto de bisimulación es una de las relaciones de invarianza estructural más importantes dentro de las lógicas modales. En particular, no sólo las fórmulas modales son invariantes bajo bisimulación, sino que existen una serie de propiedades muy interesantes que sugieren que las bisimulaciones son una respuesta adecuada a la pregunta de qué significa que dos modelos sean estructuralmente *el mismo* en el contexto modal.

Definición 2.6. *Sean $\mathcal{M} = \langle W, R, V\rangle$ y $\mathcal{M}' = \langle W', R', V'\rangle$ dos modelos. Una relación binaria no vacía $Z \subseteq W \times W$ será llamada una bisimulación entre $\mathcal{M}$ y $\mathcal{M}'$ si las siguientes condiciones son satisfechas:*

1. *(Armonía Atómica) Si wZw' entonces para todo $p \in \mathtt{PROP}$ se cumple que $V(p) = V'(p)$.*
2. *(Zig) Si wZw' y Rwv, entonces existe un $v' \in \mathcal{W}'$ tal que vZv' y $R'w'v'$.*
3. *(Zag) Si wZw' y $R'w'v'$, entonces existe un $v \in \mathcal{W}$ tal que vZv' y Rwv.*

Si $\mathcal{M}$ y $\mathcal{M}'$ están relacionados por alguna bisimulación escribiremos $\mathcal{M} \leftrightarroweq \mathcal{M}'$. Si Z es una bisimulación tal que wZw' escribiremos $Z : \mathcal{M}, w \leftrightarroweq \mathcal{M}', w'$. Cuando exista alguna bisimulación Z tal que $Z : \mathcal{M}, w \leftrightarroweq \mathcal{M}', w'$ simplemente escribiremos $\mathcal{M}, w \leftrightarroweq \mathcal{M}', w'$.

Ejemplo 2.8. Sean $\mathcal{M} = \langle W, R, V \rangle$ y $\mathcal{M}' = \langle W', R', V' \rangle$ los modelos de Kripke de la Figura 2.6 con $V(p) = V'(p) = \varnothing$ para todo $p \in \mathtt{PROP}$. Sea $Z = \{(w, w'), (w, v')\}$ como se muestra en la figura. Z es una bisimulación puesto que satisface:

1. (Armonía Atómica) Trivial puesto que $V(p) = V'(p) = \varnothing$ para todo $p \in \mathtt{PROP}$.

2. (Zig) Pensemos qué significa esta condición. La definición nos dice que siempre que un punto $w \in W$ esté relacionado (a través de Z) con otro punto $w' \in W'$ y w tenga un sucesor v, entonces w' tiene que tener un sucesor v' que esté relacionado a través de Z con v. En este ejemplo, el punto $w \in W$ sólo tiene como sucesor a w y sólo está relacionado a través de Z con w' y con v'. En primer lugar hay que ver si existe un sucesor de w' que esté relacionado con w. En segundo lugar, hay que ver si existe un sucesor de v' que esté relacionado con w. Ambos claramente existen y son v' y w', respectivamente.

3. (Zag) Notemos que esta condición es la misma que Zig, pero desde el otro modelo. En este ejemplo, el punto $w' \in W$ sólo tiene como sucesor a v' y sólo está relacionado a través de Z con w. Por otra parte el punto $v' \in W$ sólo tiene como sucesor a w' y sólo está relacionado a través de Z con w. En primer lugar hay que ver si existe un sucesor de w que esté relacionado con w'. En segundo lugar, hay que ver si existe un sucesor de w que esté relacionado con v'. Ambos claramente existen y son ambos w.

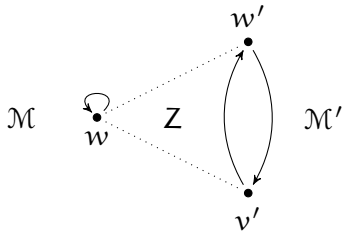

Figura 2.6: Una bisimulación Z entre los modelos $\mathcal{M}$ y $\mathcal{M}'$

El concepto de bisimulación que introducimos en esta sección es una invarianza estructural. Decimos que una propiedad es invariante bajo cierta relación u operación si, siempre que dos estructuras estén relacionadas por esa relación u operación, se cumple que una de las estructuras tiene la propiedad si y sólo si la otra la tiene. En particular, como expresamos anteriormente, las fórmulas modales son invariantes bajo bisimulación. Esto quiere decir que siempre que dos modelos sean bisimilares serán equivalentes modalmente. Este resultado es el que demostramos a continuación.

Teorema 2.1. *Sean $\mathcal{M} = \langle W, R, V \rangle$ y $\mathcal{M}' = \langle W', R', V' \rangle$ dos modelos. Sea Z una relación entre W y W'. Si $Z : \mathcal{M}, w \leftrightarroweq \mathcal{M}', w'$ entonces vale que $\mathcal{M}, w \equiv_{\mathcal{ML}} \mathcal{M}', w'$.*

Demostración. Supongamos que Z es una bisimulación entre $\mathcal{M}$ y $\mathcal{M}'$. Veamos que si $w\mathrm{Z}w'$ entonces $\mathcal{M}, w \equiv_{\mathcal{ML}} \mathcal{M}', w'$. Probemos esto por inducción estructural en las fórmulas. La proposición que queremos demostrar por inducción en las fórmulas es:

Para todo $w \in W$ y para todo $w' \in W'$, si $w\mathrm{Z}w'$ entonces $\mathcal{M}, w \equiv_{\mathcal{ML}} \mathcal{M}', w'$.

▷ Caso Base $\varphi = p \in \mathtt{PROP}$
Supongamos que $w\mathrm{Z}w'$ y que $\mathcal{M}, w \models p$. Ahora demostremos que $\mathcal{M}', w' \models p$. Por definición de $\models$ sabemos que $w \in V(p)$. Como $w\mathrm{Z}w'$ y Z es una bisimulación, por armonía atómica sabemos que $w' \in V'(p)$, pero entonces $\mathcal{M}', w' \models p$, como queríamos ver. La otra dirección es análoga.

▷ Caso Base $\varphi = \bot$
Trivial.

▷ Caso Inductivo $\varphi = \psi_1 \to \psi_2$
Supongamos $\mathcal{M}, w \models \psi_1 \to \psi_2$. Por definición de $\models$, $\mathcal{M}, w \not\models \psi_1$ o $\mathcal{M}, w \models \psi_2$. Si $\mathcal{M}, w \not\models \psi_1$, por hipótesis inductiva podemos concluír que $\mathcal{M}', w' \not\models \psi_1$, mientras que si $\mathcal{M}, w \models \psi_2$, por hipótesis inductiva podemos concluír que $\mathcal{M}', w' \models \psi_2$. En cualquiera de los dos casos concluímos que $\mathcal{M}', w' \not\models \psi_1$ o $\mathcal{M}', w' \models \psi_2$, que por definición de $\models$ es equivalente a $\mathcal{M}', w' \models \psi_1 \to \psi_2$, lo que queríamos demostrar. La otra dirección es análoga.

▷ Caso Inductivo $\varphi = \Diamond\psi$
Supongamos que $w\mathrm{Z}w'$ y que $\mathcal{M}, w \models \Diamond\psi$. Por definición de $\models$ sabemos que existe un $v \in W$ tal que $\mathrm{R}wv$ y $\mathcal{M}, v \models \varphi$. Como $w\mathrm{Z}w'$ y $\mathrm{R}wv$, por la definición de bisimulación (zig) sabemos que existe un $v' \in M'$ tal que $v\mathrm{Z}v'$ y $\mathrm{R}'w'v'$. Pero entonces por hipótesis inductiva sabemos que $\mathcal{M}', v' \models \varphi$. Por definición de $\models$ tenemos que $\mathcal{M}', w' \models \Diamond\varphi$, como queríamos demostrar. La otra dirección es análoga.

□

El concepto de bisimulación en el contexto de las lógicas modales y de las ciencias de la computación en general ha producido resultados importantes y en particular ha sido utilizado para investigar la expresividad de las lógicas modales. Entre estos resultados, destacamos el teorema de caracterización de van Benthem y el teorema de Hennessy-Milner [11]. El teorema de caracterización de van Benthem expresa que, desde una perspectiva semántica, las lógicas modales son el fragmento de primer orden invariante bajo bisimulación. El teorema de Hennessy-Milner, por otra parte, está relacionado a la pregunta de si la conversa del teorema de invarianza bajo bisimulación es válida. Es decir, si dos modelos son equivalentes modalmente ¿eso implica que son bisimilares? Si bien la respuesta para el caso general es negativa, el teorema de Hennessy-Milner nos dice que existen ciertos modelos para los cuales la conversa es verdadera. A continuación enunciamos el teorema de Hennessy-Milner. Si el lector está interesado en un enunciado más preciso del teorema de caracterización de van Benthem, o en otros resultados relacionados con el concepto de bisimulación puede consultar [11].

Definición 2.7. *Decimos que un modelo $\mathcal{M} = \langle W, R, V \rangle$ tiene imagen finita si para todo $w \in W$ el conjunto de sus sucesores $\{v \in W \mid wRv\}$ es finito.*

Teorema 2.2. *Sean $\mathcal{M} = \langle W, R, V \rangle$ y $\mathcal{M}' = \langle W', R', V' \rangle$ dos modelos con imagen finita. Entonces para todo $w \in W$ y para todo $w' \in W'$, $\mathcal{M}, w \equiv_{\mathcal{ML}} \mathcal{M}', w'$ si y sólo si $\mathcal{M}, w \leftrightarrows \mathcal{M}', w'$*

Demostración. Ver [11]. □

3 LÓGICA MODAL DINÁMICA

En este capítulo introducimos las lógicas modales dinámicas estudiadas en [1-6, 15]. Estas lógicas dinámicas, incorporan a las lógicas modales básicas operadores que cambian la relación de accesibilidad de un modelo durante la evaluación de una fórmula, por lo que las nociones de satisfacibilidad y bisimulación difieren a las de las lógicas modales básicas. Introducimos tanto la sintaxis de este lenguaje, como algunas definiciones semánticas que utilizaremos luego para poder enunciar la versión del teorema de invariancia bajo bisimulación de estas lógicas. Al final del capítulo presentamos una demostración de dicho teorema basada en la demostración dada en [3].

3.1 EL LENGUAJE DE LA LÓGICA MODAL DINÁMICA

Definición 3.1. *Sea* PROP *un conjunto infinito numerable de símbolos proposicionales, cuyos elementos denotaremos con* $p_0, p_1, \ldots$ *y un conjunto* DYN *de operadores dinámicos, definimos el lenguaje* $\mathcal{ML}(\mathtt{DYN})$ *sobre* PROP *como:*

$$\varphi ::= p \mid \bot \mid \varphi \to \varphi \mid \Diamond\varphi \mid \blacklozenge_i \varphi$$

donde $p \in \mathtt{PROP}$ *y* $\blacklozenge_i \in \mathtt{DYN}$.

Esta definición nos dice que toda fórmula o bien es un símbolo proposicional, la constante proposicional falso, una implicación, una fórmula con un diamante prefijo o bien es una fórmula con un diamante dinámico prefijo.

Notación 3.1. *Dadas dos fórmulas* $\varphi, \psi \in \mathcal{ML}(\mathtt{DYN})$ *definimos como abreviación las siguientes expresiones:*

$$\begin{aligned}
\neg\varphi &:= \varphi \to \bot \\
\top &:= \neg\bot \\
\varphi \wedge \psi &:= \neg(\varphi \to \neg\psi) \\
\varphi \vee \psi &:= \neg\varphi \to \psi \\
\varphi \leftrightarrow \psi &:= (\varphi \to \psi) \wedge (\psi \to \varphi) \\
\Box\varphi &:= \neg\Diamond\neg\varphi \\
\blacksquare_i \varphi &:= \neg\blacklozenge_i\neg\varphi
\end{aligned}$$

donde $\blacklozenge_i$ *es un operador dinámico de* DYN.

Denotaremos con $\mathcal{ML}$ *al lenguaje* $\mathcal{ML}(\{\})$. *Notar que este lenguaje es el mismo que el de la lógica modal básica. Si el conjunto* DYN *tiene un único elemento en vez de escribir* $\mathcal{ML}(\{\blacklozenge\})$ *simplemente escribiremos* $\mathcal{ML}(\blacklozenge)$.

A través de esta abreviación agregamos los operadores de la lógica proposicional, un nuevo operador modal $\Box$, el dual de $\Diamond$ y nuevos operadores dinámicos $\blacksquare_i$, duales de $\blacklozenge_i \in \mathtt{DYN}$.

3.2 FUNCIONES DE ACTUALIZACIÓN DE MODELOS

Definición 3.2. *Dado un dominio W, una función de actualización de modelos (*model update function*) para W es una función $f_W : W \times 2^{W^2} \to 2^{W \times 2^{W^2}}$ que toma un estado en W y una relación binaria sobre W y devuelve un conjunto de posibles actualizaciones al estado de evaluación y la relación de accesibilidad.*

Definición 3.3. *Dada $\mathcal{C}$ una clase de modelos, una familia de funciones de actualización de modelos $\mathtt{f}$ es una clase de funciones de actualización de modelos tal que para cada dominio de $\mathcal{C}$ existe una única función de actualización de modelos en $\mathtt{f}$:*

$$\mathtt{f} = \{f_W \mid \langle W, R, V\rangle \in \mathcal{C}\}$$

Definición 3.4. *Decimos que una clase de modelos $\mathcal{C}$ es cerrada bajo una familia de funciones de actualización de modelos $\mathtt{f}$ si para todo modelo $\mathcal{M} = \langle W, R, V\rangle$ de $\mathcal{C}$ se cumple:*

$$\{\langle W, R', V\rangle \mid f_W \in \mathtt{f}, w \in W, (v, R') \in f_W(w, R)\} \subseteq \mathcal{C}$$

La clase de todos los modelos es cerrada bajo cualquier familia de funciones de actualización de modelos. En el resto del trabajo asumiremos que DYN es un conjunto de operadores dinámicos, que $\mathcal{C}$ es la clase de todos los modelos y que F asocia a cada elemento de DYN una familia de funciones de actualización de modelos. Si $\blacklozenge_i \in$ DYN denotaremos con $F(\blacklozenge_i)$ a la familia de funciones de actualización de modelos asociada a $\blacklozenge_i$ por F.

3.3 SATISFACIBILIDAD

A continuación introducimos la definición de satisfacibilidad para estas lógicas modales dinámicas. Este concepto determinará el significado que le daremos a las fórmulas del lenguaje modal dinámico. Notemos que los modelos sobre los cuales daremos esta definición serán los mismos que los modelos de las lógicas modales básicas, es decir, modelos de Kripke.

Definición 3.5. *Sean $\mathcal{M} = \langle W, R, V\rangle$ un modelo, $w \in W$ un punto y $\varphi \in \mathcal{ML}(\mathtt{DYN})$ una fórmula. Diremos que $\mathcal{M}, w$ satisfacen φ y escribiremos $\mathcal{M}, w \models \varphi$, si:*

$$\begin{aligned}
\mathcal{M}, w &\models p \text{ sii } w \in V(p)\\
\mathcal{M}, w &\models \bot \text{ sii nunca}\\
\mathcal{M}, w &\models \varphi \to \psi \text{ sii no se cumple } \mathcal{M}, w \models \varphi \text{ o se cumple } \mathcal{M}, w \models \psi\\
\mathcal{M}, w &\models \Diamond\varphi \text{ sii existe un } v \in W \text{ t.q. } Rwv \text{ y } \mathcal{M}, v \models \varphi\\
\mathcal{M}, w &\models \blacklozenge_i\varphi \text{ sii existe un } (v, R') \in f_W(w, R) \text{ t.q. } \langle W, R', V\rangle, v \models \varphi\\
&\qquad \text{donde } \mathtt{f} = F(\blacklozenge_i)
\end{aligned}$$

Siempre que no ocurra $\mathcal{M}, w \models \varphi$ escribiremos $\mathcal{M}, w \not\models \varphi$.

Notemos que esta definición de satisfacibilidad modal dinámica es una extensión de la definición de satisfacibilidad modal que dimos en el capítulo anterior.

La diferencia está en que ahora nuestro lenguaje puede expresar cambios en la relación de accesibilidad. Por ejemplo si definimos un operador modal dinámico que elimine una arista arbitraria de la relación de accesibilidad, podremos expresar este cambio de la relación de accesibilidad en este lenguaje modal dinámico. A continuación definimos la función de actualización de modelos correspondiente a este operador dinámico conocido como el operador de sabotaje de van Benthem. Además definimos otras cinco funciones de actualización de modelos que darán lugar a otros cinco operadores dinámicos. Luego presentaremos algunos ejemplos utilizando algunos de estos operadores.

Notación 3.2. *Escribiremos wv en vez que $\{(w,v)\}$ o (w,v). Siempre que usemos esta notación se podrá desambiguar a cuál de las dos nos referimos. Dada una relación binaria* R *definimos la siguiente notación:*

$$R^{-}_{wv} = R \setminus wv$$
$$R^{+}_{wv} = R \cup wv$$
$$R^{*}_{wv} = (R \setminus vw) \cup wv$$

A continuación definimos las seis funciones de actualización de modelos que darán lugar a los siguientes seis operadores modales dinámicos:

- El operador de sabotaje de van Benthem $\blacklozenge_{gsb}$.
- Una versión local del operador de sabotaje de van Benthem $\blacklozenge_{sb}$ que elimina una arista existente entre el estado actual de evaluación y un estado sucesor, el cual se convierte el nuevo punto de evaluación.
- Un operador de puente $\blacklozenge_{gbr}$ que agrega una arista entre dos estados desconectados.
- Una versión local del operador de puente $\blacklozenge_{br}$ que agrega una arista entre el estado actual de evaluación y un estado inalcanzable, el cual se convierte el nuevo punto de evaluación.
- Un operador de intercambio $\blacklozenge_{gsw}$ que invierte la dirección de una arista.
- Una versión local del operador de intercambio $\blacklozenge_{sw}$ que cambia la dirección de una arista entre el estado actual de evaluación y un estado sucesor, el cual se convierte el nuevo punto de evaluación.

Sea W un dominio y R una relación binaria sobre W,

- $f^{gsb}_{W}(w,R) = \{(w, R^{-}_{uv}) \mid uv \in R\}$
- $f^{sb}_{W}(w,R) = \{(v, R^{-}_{wv}) \mid wv \in R\}$
- $f^{gbr}_{W}(w,R) = \{(w, R^{+}_{uv}) \mid uv \notin R\}$
- $f^{br}_{W}(w,R) = \{(v, R^{+}_{wv}) \mid wv \notin R\}$
- $f^{gsw}_{W}(w,R) = \{(w, R^{*}_{uv}) \mid uv \in R\}$
- $f^{sw}_{W}(w,R) = \{(v, R^{*}_{wv}) \mid wv \in R\}$

Ejemplo 3.1. Sean $\mathcal{M} = \langle W, R, V\rangle$ y $\mathcal{M}' = \langle W', R', V'\rangle$ los modelos de Kripke de la Figura 3.1 con $V(p) = V'(p) = \varnothing$ para todo $p \in \texttt{PROP}$. Sea $\texttt{DYN} = \{\blacklozenge_{sb}\}$ el conjunto de operadores dinámicos.

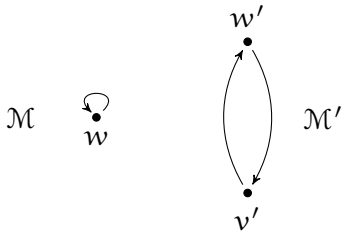

Figura 3.1: Los modelos de Kripke $\mathcal{M}$, $\mathcal{M}'$

Tomemos la fórmula $\varphi = \blacklozenge_{sb}\Diamond\top$ del lenguaje $\mathcal{ML}(\texttt{DYN})$. Notemos que φ no es válida en el *pointed model* $(\mathcal{M}, w)$: si bien existe un estado sucesor, una vez que nos movemos a este estado, la arista deja de existir en la relación de accesibilidad y como $\langle W, R^{-}_{ww}, V\rangle, w \not\models \Diamond\top$ es claro que $\mathcal{M}, w \not\models \varphi$. Pero en el *pointed model* $(\mathcal{M}', w')$ sí es válida: una vez que nos movemos al estado sucesor v' podemos volver al estado w'. Por lo que $\mathcal{M}', w' \models \varphi$. En el Ejemplo 2.8 del capítulo anterior vimos que estos dos mismos modelos no podían ser distinguidos por la lógica modal básica; aquí mostramos cómo el lenguaje modal dinámico $\mathcal{ML}\{\blacklozenge_{sb}\}$ sí puede distinguirlos.

Ejemplo 3.2. Sea $\texttt{DYN} = \{\blacklozenge_{gsb}\}$ el conjunto de operadores dinámicos ¿Para qué *pointed models* se satisface la fórmula $\varphi = \Diamond\Diamond\top \wedge \blacksquare_{gsb}\Box\bot$ del lenguaje $\mathcal{ML}(\texttt{DYN})$? Notemos que la fórmula $\blacksquare_{gsb}\Box\bot$ significa que cualquier sabotaje global hace que el punto de evaluación no tenga vecinos, mientras que la fórmula $\Diamond\Diamond\top$ significa que existe un vecino a dos saltos del punto de evaluación. Ambas condiciones sólo son posibles si el punto que estamos evaluando es reflexivo y si no existe ninguna otra arista en la relación de accesibilidad. Esta propiedad no es expresable en lógica modal básica.

3.4 BISIMULACIONES

Los ejemplos de la sección anterior muestran que las lógicas modales dinámicas son más expresivas que las lógicas modales básicas, sin embargo, todavía falta dar una definición de bisimulación para estas lógicas modales dinámicas. En esta sección introducimos la noción de equivalencia modal dinámica y una definición de bisimulación que es apropiada para estas lógicas modales dinámicas. Finalmente demostramos el teorema de invariancia de las fórmulas modales dinámicas bajo bisimulación.

Definición 3.6. *Sean $\mathcal{M} = \langle W, R, V\rangle$ y $\mathcal{M}' = \langle W', R', V'\rangle$ dos modelos, $w \in W$ y $w' \in W'$ dos puntos. Diremos que $\mathcal{M}, w$ y $\mathcal{M}', w'$ satisfacen las mismas fórmulas de $\mathcal{ML}(\texttt{DYN})$ y escribiremos $\mathcal{M}, w \equiv_{\mathcal{ML}(\texttt{DYN})} \mathcal{M}', w'$ si se cumple que para toda $\varphi \in \mathcal{ML}(\texttt{DYN})$:*

$$\mathcal{M}, w \models \varphi \textit{ sii } \mathcal{M}', w' \models \varphi'$$

Definición 3.7. *Sean $\mathcal{M} = \langle W, R, V\rangle$ y $\mathcal{M}' = \langle W', R', V'\rangle$ dos modelos. Diremos que una relación binaria no vacía $Z \subseteq (W \times 2^{W^2}) \times (W' \times 2^{W'^2})$ es una bisimulación entre $\mathcal{M}$ y $\mathcal{M}'$ si asumiendo que $(w, S)Z(w', S')$ podemos probar que:*

1. *(Armonía Atómica) Para todo $p \in$ **PROP** se cumple que $L(w, p) = L'(w', p)$.*
2. *(Zig) Si Swv, entonces existe un $v' \in W'$ tal que $(v, S)Z(v', S')$ y $R'w'v'$.*
3. *(Zag) Si $S'w'v'$, entonces existe un $v \in W$ tal que $(v, S)Z(v', S')$ y Rwv.*
4. *(f-Zig) Si para toda $\blacklozenge_i \in$ **DYN** se cumple que si $(v, T) \in f_W(w, S)$, entonces existe un $(v', T') \in f_{W'}(w', S')$ t.q. $(v, T)Z(v', T')$ donde $f = F(\blacklozenge_i)$.*
5. *(f-Zag) Si para toda $\blacklozenge_i \in$ **DYN** se cumple que si $(v', T') \in f_{W'}(w', S')$, entonces existe un $(v, T) \in f_W(w, S)$ t.q. $(v, T)Z(v', T')$ donde $f = F(\blacklozenge_i)$.*

Si $\mathcal{M}$ y $\mathcal{M}'$ están relacionados por alguna bisimulación escribiremos $\mathcal{M} \leftrightarrows_{\mathcal{ML}(DYN)} \mathcal{M}'$. Si Z es una bisimulación tal que $(w, R)Z(w', R')$ escribiremos $Z : \mathcal{M}, w \leftrightarrows_{\mathcal{ML}(DYN)} \mathcal{M}', w'$ donde R y R' son las relaciones de $\mathcal{M}$ y $\mathcal{M}'$ respectivamente. Cuando exista alguna bisimulación Z tal que $Z : \mathcal{M}, w \leftrightarrows_{\mathcal{ML}(DYN)} \mathcal{M}', w'$ simplemente escribiremos $\mathcal{M}, w \leftrightarrows_{\mathcal{ML}(DYN)} \mathcal{M}', w'$.

Equipados con esta noción de bisimulación podemos probar el siguiente teorema de invarianza, demostrado en [3].

Teorema 3.1. *Sean $\langle W, S, V\rangle$, $\langle W', S', V'\rangle$ dos modelos de Kripke y $w \in W$ y $w' \in W'$. Si $\langle W, S, V\rangle, w \leftrightarrows_{\mathcal{ML}(DYN)} \langle W', S', V'\rangle, w'$ entonces $\langle W, S, V\rangle, w \equiv_{\mathcal{ML}(DYN)} \langle W', S', V'\rangle, w'$.*

Demostración. Probemos esto por inducción estructural en las fórmulas.

Antes de continuar introducimos la siguiente notación: Sea T una relación binaria sobre W, denotaremos con $\mathcal{M}_T$ a $\langle W, T, V\rangle$. Sea T una relación binaria sobre W', denotaremos con $\mathcal{M}'_T$ a $\langle W', T, V'\rangle$.

La proposición que queremos demostrar por inducción en las fórmulas es:

$$\begin{gathered}\text{Para todo } w \in W,\ w' \in W',\ S \subseteq W \times W,\ S' \subseteq W' \times W',\\ \text{y para todo } Z \in (W \times 2^{W^2}) \times (W' \times 2^{W'^2}),\\ \text{si } Z : \mathcal{M}_S, w \leftrightarrows_{\mathcal{ML}(DYN)} \mathcal{M}'_{S'}, w' \text{ entonces } \mathcal{M}_S, w \equiv_{\mathcal{ML}(DYN)} \mathcal{M}'_{S'}, w'.\end{gathered}$$

▷ Caso Base $\varphi = p \in$ PROP

Supongamos que $(w, S)Z(w', S')$ y que $\mathcal{M}_S, w \models p$. Ahora demostremos que $\mathcal{M}'_{S'}, w' \models p$. Por definición de $\models$ sabemos que $w \in V(p)$. Como $(w, S)Z(w', S')$ y Z es una bisimulación, por armonía atómica sabemos que $V'(w', p)$, pero entonces $\mathcal{M}'_{S'}, w' \models p$, como queríamos ver. La otra dirección es análoga.

▷ Caso Base $\varphi = \bot$

Trivial.

▷ Caso Inductivo $\varphi = \psi_1 \to \psi_2$

Supongamos $\mathcal{M}_S, w \models \psi_1 \to \psi_2$. Por definición de $\models$, $\mathcal{M}_S, w \not\models \psi_1$ o $\mathcal{M}_S, w \models \psi_2$. Si $\mathcal{M}_S, w \not\models \psi_1$, por hipótesis inductiva podemos concluír que $\mathcal{M}'_{S'}, w' \not\models \psi_1$, mientras que si $\mathcal{M}_S, w \models \psi_2$, por hipótesis inductiva podemos concluír que $\mathcal{M}'_{S'}, w' \models \psi_2$. En cualquiera de los dos casos concluímos que $\mathcal{M}'_{S'}, w' \not\models \psi_1$ o

$\mathcal{M}'_{S'}, w' \models \psi_2$, que por definición de $\models$ es equivalente a $\mathcal{M}'_{S'}, w' \models \psi_1 \to \psi_2$, lo que queríamos demostrar. La otra dirección es análoga.

▷ Caso Inductivo $\varphi = \Diamond\psi$

Supongamos que $(w, S)Z(w', S')$ y que $\mathcal{M}_S, w \models \Diamond\psi$. Por definición de $\models$ sabemos que existe un $v \in W$ tal que Swv y $\mathcal{M}_S, v \models \varphi$. Como $(w, S)Z(w', S')$ y Swv, por la definición de bisimulación (zig) sabemos que existe un $v' \in M'$ tal que $(v, S)Z(v', S')$ y $S'w'v'$. Pero entonces por hipótesis inductiva sabemos que $\mathcal{M}'_{S'}, v' \models \varphi$. Por definición de $\models$ tenemos que $\mathcal{M}'_{S'}, w' \models \Diamond\varphi$, como queríamos demostrar. La otra dirección es análoga.

▷ Caso Inductivo $\varphi = \blacklozenge_i\psi$

Sea $f = F(\blacklozenge_i)$. Supongamos que $Z : \mathcal{M}_S, w \leftrightarrows_{\mathcal{ML}(\mathtt{DYN})} \mathcal{M}'_{S'}, w'$ y que $\mathcal{M}_S, w \models \blacklozenge\psi$. Por definición de $\models$ sabemos que existen $(v, T) \in f_W(w, S)$ tales que $\mathcal{M}_T, v \models \varphi$. Como $(w, S)Z(w', S')$ y $(v, T) \in f_W(w, S)$, por la definición de bisimulación (f-zig) sabemos que existen $(v', T') \in f_{W'}(w', S')$ tales que $(v, T)Z(v', T')$. Pero entonces por la hipótesis inductiva sabemos que:

$$\begin{gathered}\text{Para todo } Z' \in (W \times 2^{W^2}) \times (W' \times 2^{W'^2}),\\ \text{si } Z' : \mathcal{M}_T, v \leftrightarrows_{\mathcal{ML}(\mathtt{DYN})} \mathcal{M}'_{T'}, v' \text{ entonces } \mathcal{M}_T, v \equiv_{\mathcal{ML}(\mathtt{DYN})} \mathcal{M}'_{T'}, v'.\end{gathered}$$

Como $(v, T)Z(v', T')$ y Z es una bisimulación entre $\mathcal{M}_T$ y $\mathcal{M}'_{T'}$ sabemos que $Z : \mathcal{M}_T, v \leftrightarrows_{\mathcal{ML}(\mathtt{DYN})} \mathcal{M}'_{T'}, v'$. Luego $\mathcal{M}_T, v \equiv_{\mathcal{ML}(\mathtt{DYN})} \mathcal{M}'_{T'}, v'$. Como sabemos que $\mathcal{M}_T, v \models \varphi$, obtenemos que $\mathcal{M}'_{S'}, w' \models \blacklozenge\varphi$, como queríamos demostrar.

La otra dirección es análoga.

□

Como vimos en el capítulo anterior, en las lógicas modales básicas no sólo vale el teorema de invarianza, sino que también es válida una versión restringida de la conversa. Este resultado, conocido como Teorema de Henessy-Milner, tiene un análogo para las lógicas modales dinámicas presentadas aquí. La demostración de este teorema y otros de naturaleza similar puede encontrarse en [3].

4 COQ

En este capítulo introducimos las características más relevantes del asistente de demostraciones Coq. Si bien este capítulo no constituye un tutorial completo, el mismo tiene el objetivo de introducir al lector en las herramientas de Coq que fueron utilizadas para formalizar las lógicas modales presentadas previamente.

Para una referencia completa de Coq visitar el manual de referencia en https://coq.inria.fr/refman.

4.1 COQ COMO UN LENGUAJE FUNCIONAL

Si bien Coq es un asistente de demostraciones, Coq tambíen puede ser considerado un lenguaje funcional muy similar a lenguajes como OCaml y Haskell. Decimos que es un lenguaje funcional pues el cómputo en Coq es logrado a través de la evaluación de funciones puras. Las funciones puras siempre devuelven el mismo valor al ser aplicadas a un determinado argumento y nunca producen efectos secundarios, es decir que su aplicación no modifica el estado del entorno. Estas propiedades de las funciones puras hacen que razonar tanto formal como informalmente sobre los programas funcionales de Coq sea más sencillo.

Como muchos otros lenguajes funcionales, Coq permite trabajar con funciones como si fueran cualquier otro tipo de dato. En particular las funciones pueden estar presentes en estructuras de datos y pueden tomar como entrada otras funciones y devolver como salida otras funciones.

Coq es un lenguaje muy seguro respecto a los tipos. En particular puede detectar estáticamente si una función está siendo aplicada a los tipos correctos o no. Por ejemplo, si una función tiene como dominio a los naturales y es aplicada a un argumento que no tiene tipo natural, Coq puede detectar el error estáticamente.

Otra característica de Coq es la capacidad de definir tipos inductivos y realizar *pattern matching* sobre los mismos. Estas dos herramientas, como veremos más adelante, nos permitirán definir y manipular estructuras de datos de maneras muy elegantes.

4.2 DEFINIENDO NUEVOS TIPOS DE DATOS Y FUNCIONES

Coq nos permite definir nuevos tipos de datos a través del comando `Inductive`. Supongamos que quisiéramos definir el tipo de los booleanos que llamaremos `bool`. Notemos que existen dos habitantes de este tipo: `true` y `false`. Esto puede ser formalizado en Coq de la siguiente manera:

```
Inductive bool :=
```

```
| true : bool
| false : bool.
```

`true` y `false` son denominados los constructores del tipo `bool`. Los constructores, como veremos luego, no necesariamente son habitantes del tipo, también pueden ser funciones que toman ciertos argumentos y permiten construir nuevos habitantes del tipo a partir de estos argumentos.

Estos tipos de datos no serían de mucha utilidad si Coq no nos permitiera definir funciones sobre ellos. Para definir una función no recursiva en Coq podemos hacer:

```
Definition not (p : bool) :=
match p with
| true ⇒ false
| false ⇒ true
end.

Definition or (p : bool) (q :  bool) :=
match p with
| true ⇒ true
| false ⇒ q
end.
```

Lo primero que escribimos luego de `Definition` es el nombre de la función, luego siguen los argumentos con sus tipos y finalmente, luego del símbolo := la definición de la función. `match p with ... end` es la sintaxis que provee Coq para poder realizar *pattern matching*. Los distintos patrones a buscar se separan con el símbolo `|`. En el caso de los booleanos sólo hay dos posibilidades puesto que sólo existen dos constructores: `true` y `false`. Luego del símbolo ⇒ en cada constructor, se especifica el valor que se debe retornar cuando `p` haya coincidido con el constructor.

Si quisiéramos aplicar esta función a `true` y al resultado de aplicar `not` a `true` bastaría con escribir `or true (not true)`. Notar que la sintaxis es la misma sintaxis aplicativa prefija de otros lenguajes funcionales como Haskell y OCaml. Para computar el resultado Coq provee el comando `Compute`:

```
Compute or true (not true).
```

Coq responde a esta consulta con:

```
= false : bool
```

Como decíamos antes, Coq posee un sistema de tipos muy seguro y no nos permite aplicar la función a valores con tipos incorrectos. Por ejemplo, si quisiéramos aplicar la función `or` a un número natural Coq no nos lo permite puesto que el tipo de `or` es `bool → bool → bool`. ¿Qué significa esta notación? Dados dos tipos `A` y `B`, `A → B` es el tipo de una función que toma elementos en `A` y retorna elementos en `B`. Se asume que → es asociativa a la derecha, por lo que `A → B → C` debe leerse como `A → (B → C)`. Por este motivo, `bool → bool → bool` es el tipo de las funciones que toman un booleano y devuelven funciones que toman un booleano y retornan un booleano. Claramente esta función no puede ser aplicada a un valor que no sea o bien `true` o bien `false`.

4.3 TIPOS DE DATOS CON CONSTRUCTORES PARAMÉTRICOS

Como mencionamos antes, Coq nos permite definir un tipo dando constructores que toman argumentos de algún tipo y a partir de estos valores construyen un nuevo habitante del tipo. Veamos un ejemplo:

```
Inductive BooleanPair :=
  P : bool → bool → BooleanPair.
```

El tipo `BooleanPair` claramente representa el tipo de todos los pares de booleanos. Notemos que el único constructor de `BooleanPair` toma dos booleanos y contruye un `BooleanPair` a partir de estos.

Los argumentos de los constructores pueden tener el mismo tipo que el que se está definiendo. Los tipos que poseen estos constructores se conocen como tipos de datos recursivos. A modo de ejemplo, supongamos que quisiéramos definir los números naturales. Un primer intento ingenuo para hacer esto sería:

```
Inductive nat :=
  | Zero : nat
  | One : nat
  | Two : nat
  | Three : nat
  | Four : nat
  | Five : nat.
```

Pero esto claramente no es una buena idea. Si recordamos que todo número natural o bien es cero o bien es el sucesor de algún número natural podemos formalizar los números naturales en Coq de la siguiente manera:

```
Inductive nat :=
  | Zero : nat
  | Succ : nat → nat.
```

Para construir un número natural arbitrario n basta con aplicar `Succ` n veces a `Zero`. Por ejemplo 5 se puede construir como:

```
Succ (Succ (Succ (Succ (Succ Zero))))
```

El uso de *pattern matching* sobre estos tipos de datos con constructores más complejos es muy sencillo. Para ver esto, supongamos que queremos definir una función f de la forma:

$$f(0) = 0$$
$$f(n+1) = n$$

para todo número natural n usando el tipo `nat`.

Para ello debemos considerar las dos formas en las que pudo haber sido construída la entrada. O bien la entrada a la función fue obtenida usando el constructor `Zero` o bien fue obtenida usando `Succ`:

```
Definition minusOne (n : nat) :=
match n with
| Zero ⇒ Zero
| Succ n' ⇒ n'
end.
```

Es evidente que esto implementa correctamente la función f.

4.4 FUNCIONES RECURSIVAS

Supongamos ahora que quisiéramos definir la suma entre dos números naturales. Recordemos que:

$$0 + m = m$$
$$(n + 1) + m = (m + n) + 1$$

Por lo que sería natural intentar hacer:

```
Definition plus (n : nat) (m : nat) :=
match n with
| Zero ⇒ m
| Succ n' ⇒ Succ (plus n' m)
end.
```

Pero Coq no permite usar `Definition` para dar definiciones recursivas. Para asegurar que `plus` está bien definida la definición no puede contener aquello que se está definiendo. Para resolver esto, Coq nos permite utilizar un operador de punto fijo llamado `Fixpoint` que realiza una recursión estructural sobre el parámetro `n`:

```
Fixpoint plus (n : nat) (m : nat) :=
match n with
| Zero ⇒ m
| Succ n' ⇒ Succ (plus n' m)
end.
```

No cualquier función puede ser definida en Coq. Esto se debe a que el sistema de Coq requiere que todas las funciones terminen. A modo de ejemplo, la siguiente función no es aceptada:

```
Fixpoint loop (n : nat) := loop n
```

Como el sistema de Coq no puede determinar si un programa arbitrario va a terminar o no, cada definición recursiva debe ser decreciente en alguno de los argumentos. Esto quiere decir que uno de los argumentos de la función se hace cada vez más pequeño en cada llamada recursiva. Esto asegura la terminación de la función.

A modo de ejemplo, si bien la siguiente función termina para cualquier entrada, Coq la rechaza porque no puede determinar en qué argumento es decreciente:

```
Fixpoint zero (n : nat) (m : nat) :=
  match n with
  | Zero   ⇒ match m with
             | Zero ⇒ Zero
             | Succ m' ⇒ zero Zero m'
             end
  | Succ n' ⇒ match m with
              | Zero ⇒ zero n' Zero
              | Succ m' ⇒ zero n' m'
              end
  end.
```

El requerimiento de que Coq pueda determinar la terminación de todo programa es esencial para que el sistema pueda funcionar como un asistente de demostraciones.

4.5 SORTS

En Coq todo objeto del formalismo tiene asociado un tipo, incluídos los tipos. Por ejemplo, `nat` es un tipo cuyo tipo es `Set` y `Set` es un tipo cuyo tipo es `Type`. Los tipos como `Set` y `Type` que tienen como habitantes otros tipos se los conoce como *sorts*.

Pero si todo objeto del formalismo tiene asociado un tipo, cabe preguntarse, entonces, cuál es el tipo de `Type`. Coq posee una jerarquía infinita de tipos `Type(i)` con $i \in \mathbb{N}$ en donde `Type(i)` es un habitante de `Type(i+1)` para todo $i \in \mathbb{N}$. Coq construye esta jerarquía para evitar una inconsistencia en el sistema que ocurre cuando se asume que `Type` es un habitante de `Type`.

El usuario de Coq no tiene que mencionar explícitamente el índice `i` cuando utiliza el tipo `Type`, basta con que escriba `Type`. El sistema de tipos de Coq genera por cada instancia de `Type` un nuevo índice. Es por esto, que en la práctica podemos escribir `Type : Type`

Existe otro *sort* llamado `Prop`, el tipo de las proposiciones lógicas. Este tipo nos permitirá especificar propiedades de los programas. `Prop`, como todos los otros tipos, también es un habitante de `Type`.

4.6 TIPOS POLIMÓRFICOS

Ahora volvamos al ejemplo de `BooleanPair`. Supongamos que quisiéramos definir un par de números naturales. Para ello tendríamos que hacer:

```
Inductive NaturalPair :=
  P : nat → nat → NaturalPair.
```

Si quisiéramos hacer lo mismo para otro tipo tendríamos que definir nuevamente un nuevo tipo de datos para ese tipo. Esto claramente no es conveniente. Para este tipo de situaciones, Coq provee un mecanismo que permite definir funciones que dado un tipo devuelven un nuevo tipo de datos que depende de esos tipos. Podemos pensar esto como un tipo genérico.

En nuestro caso queremos definir un tipo de pares que sea lo suficientemente general para abarcar cualquier par de objetos. En Coq es posible lograr esto de la siguiente manera:

```
Inductive pair (A : Type) (B : Type) :=
  P : A → B → pair A B.
```

Escribimos `(A : Type) (B : Type)` para indicarle a Coq que tanto `A` como `B` tienen tipo `Type`, es decir para indicarle a Coq que son dos tipos cualesquiera.

Si queremos crear un par de un natural y un booleano podemos escribir `P nat bool Zero false`. Notemos que es necesario indicarle a Coq cuáles son los tipos del par. Como esta notación es redundante en la mayoría de los casos, podemos pedirle a Coq que infiera automáticamente qué tipos tiene `Zero` y `false` y que nos permita escribir simplemente `P Zero false`. Para que Coq inferiera el tipo del par debemos definir `pair` usando la siguiente sintaxis:

```
Inductive pair {A : Type} {B : Type} :=
  P : A → B → pair.
```

4.7 TIPOS DEPENDIENTES

Supongamos que queremos definir un tipo de datos de vectores de dos dimensiones de booleanos. Si bien podemos definir todas las funciones asumiendo que tenemos un vector de dos elementos, muchas de las operaciones pueden ser generalizadas para vectores de una dimensión arbitraria. Podríamos usar una lista común, pero esto no nos permitiría verificar estáticamente el tamaño del vector. Con la forma de construir nuevos tipos de datos que vimos hasta ahora no es factible contruir el tipo de datos que queremos, pero Coq posee un poderoso sistema de tipos dependientes que nos permite lograr lo que queremos:

```
Inductive vector : nat → Type :=
| nilv : vector Zero
| consv : forall (n : nat), bool → vector n → vector (succ n).
```

El tipo de consv es un tipo dependiente. Informalmente, un tipo dependiente es simplemente un tipo cuya definición depende de un valor. El tipo del constructor `consv` es un tipo dependiente puesto que el tipo que retorna varía en función del valor `n`. El tipo `forall (n : nat), bool → vector n → vector (succ n)` debe entenderse de la siguiente manera: Dado un número natural `n`, un booleano y un vector de longitud `n`, podemos construir un vector de longitud `n + 1`.

Definamos esta noción de tipo dependiente más formalmente. Dado un tipo `T : S` perteneciente a un *sort* `S`, se puede definir una familia de tipos `F : T → S` que a cada término `t : T` le asigna un tipo `F(t) : S`. Diremos que `F` varía en función de `t`. Una función cuyo tipo de valor de retorno varía en función de sus argumentos es una función dependiente y el tipo de esta función se denomina *dependent product type*.

4.8 TEOREMAS Y DEMOSTRACIONES

Hasta ahora vimos Coq como un lenguaje de programación funcional. Pero como dijimos previamente Coq es un asistente de demostraciones y como tal es capaz de expresar propiedades de los programas en una lógica de alto orden. Además provee mecanismos para dar demostraciones de estas propiedades y verificar automáticamente la correctitud de las mismas.

Empecemos por un ejemplo simple. Supongamos que queremos demostrar la proposición:

$$\forall P, P \to P$$

Lo primero que tenemos que hacer es, usando el comando `Theorem`, declarar el teorema y asignarle un nombre:

```
Theorem PimpliesP : forall P : Prop, P → P.
```

Este nombre nos será útil para poder referirnos al teorema en otras demostraciones.

El asistente de demostraciones de Coq es interactivo, y luego de que ejecutamos el comando anterior nos muestra que:

```
forall P : Prop, P → P
```

Lo que nos está diciendo es que debemos demostrar que para cualquier `P : Prop` se satisface `P → P`. Para demostrar teoremas en Coq utilizamos *backward reasoning*. Cuando utilizamos una regla de inferencia en matemática transformamos una o varias fórmulas que llamamos premisas en una fórmula que llamamos conclusión. Es común en matemática aplicar las reglas de inferencia hacia adelante (*forward reasoning*), es decir, primero demostramos todas las premisas y una vez que hicimos esto, usamos la regla de inferencia para obtener la conclusión. Entonces, si quisieramos demostrar una fórmula `P ∧ Q`, primero probaríamos `Q` y `P` y luego, utilizando la regla de inferencia concluiríamos `P ∧ Q`. En Coq el razonamiento se hace hacia atrás (*backward reasoning*), esto es, primero aplicamos la regla de inferencia a lo que queremos demostrar y luego demostramos todas las premisas de esa regla de inferencia. Entonces, si quisieramos demostrar que `P ∧ Q`, primero aplicaríamos la regla de inferencia y luego demostraríamos `P` y `Q`.

En Coq las conclusiones se denominan objetivos (*goals*) y las premisas subobjetivos (*subgoals*). Las tácticas implementan la aplicación de reglas de inferencia con razonamiento hacia atrás. Una vez que aplicamos una táctica a un objetivo, Coq nos muestra una lista de subobjetivos a demostrar. Una vez que todos los subobjetivos fueron demostrados, el objetivo está demostrado.

Para empezar a demostrar el teorema `forall P : Prop, P → P` y usar el lenguaje de tácticas para construir una demostración tenemos que ejecutar el comando `Proof`. Lo primero que queremos hacer es suponer que tenemos una proposición `P` de tipo `Prop`. Para esto podemos ejecutar la táctica `intro`. Una vez ejecutada la táctica Coq nos responde que:

```
P : Prop
```

```
P → P
```

Coq nos está indicando que debemos demostrar `P → P` asumiendo `P : Prop`. Sobre las líneas se encuentran las hipótesis, mientras que debajo se muestra el subobjetivo (*subgoal*) a demostrar. La única hipótesis que tenemos hasta ahora es que `P` tiene tipo `Prop`.

Para continuar volvemos a ejecutar la táctica `intro`. Esta vez Coq nos muestra:

```
P : Prop
H : P
```

```
P
```

Ahora tenemos una nueva hipótesis que Coq nombró automáticamente como `H`. Si quisieramos ponerle otro nombre podríamos haber hecho `intro otroNombreParaH` en vez de `intro`.

Si bien hasta ahora interpretábamos el símbolo `:` como "tiene tipo" en este caso podemos pensar que `H : P` significa "`H` es una demostración de `P`". Por lo que lo único que debemos hacer es usar esta demostración de `P` para demostrar `P`. Para esto, usamos la táctica `exact H` que le indica a Coq que debe utilizar la prueba `H` para demostrar `P`.

Para finalizar la demostración debemos ejecutar el comando `Qed`. La demostración completa queda:

```
Proof.
    intro.
    intro.
    exact H.
Qed.
```

Este ejemplo nos muestra claramente que `H : P` puede ser leído tanto como "`H` tiene tipo `P`" como "`H` es una demostración o una evidencia de la proposición `P`". Siempre que ocurra que `P : Prop` va a resultar más natural la segunda lectura. Esta posibilidad de interpretar `:` de dos maneras distintas tiene que ver con un isomorfismo entre proposiciones lógicas y tipos, que relaciona las demostraciones de las proposiciones con habitantes de los tipos. Este isomorfismo, conocido como isomorfismo de Curry-Howard [19], es central en la teoría que subyace a Coq.

Las demostraciones, como la que construímos más arriba, son términos cuyo tipo es lo que demuestran. Por ejemplo, si en el ejemplo anterior utilizamos el comando `Show Proof.`:

```
Proof.
    intro.
    intro.
    exact H.
    Show Proof.
Qed.
```

Coq nos muestra el término de prueba `fun (P : Prop) (H : P) ⇒ H.`

De hecho, dado que las demostraciones son términos cuyo tipo es lo que demuestran, podemos demostrar algo construyendo el término explícitamente:

```
Definition PimpliesP : forall P : Prop, P → P :=
fun (P : Prop) (H : P) ⇒ H.
```

Si el verificador de tipos de Coq acepta esta definición es porque `PimpliesP` tiene tipo `forall P : Prop, P → P`, o lo que es lo mismo, que `PimpliesP` demuestra la proposición `forall P : Prop, P → P`.

El lector podría cuestionar la utilidad de tener un comando `Proof` adicional para demostrar teoremas si al fin y al cabo los términos de prueba pueden ser construídos explícitamente. Pero el propósito de usar `Proof` es que nos permite automatizar la construcción de demostraciones a través del uso de tácticas. Luego de la ejecución de `Proof`, el sistema de Coq nos permite introducir tácticas, como `intro` y `exact H` que generan los términos automáticamente.

Existen tácticas mucho más sofisticadas que las presentadas aquí. De hecho, existen lenguajes de tácticas como L_{tac} [14], el que usamos en este trabajo, y Mtac2 [21] con los que se pueden definir nuevas tácticas a partir de tácticas más sencillas. De esta forma, los usuarios pueden definir tácticas aprovechando el conocimiento que pudieran tener sobre la estructura particular de ciertas demostraciones.

4.9 TÁCTICAS

En la Sección 4.8 vimos la necesidad de disponer de tácticas para facilitar la construcción de demostraciones. En esta sección, introducimos algunas de las tácticas de L_{tac} más utilizadas.

Existenciales

En Coq, para demostrar un existencial $\exists x, P(x)$ debemos dar explícitamente un objeto x tal que $P(x)$. Para ello, disponemos de la táctica `exists`. Por ejemplo, si quisiéramos probar:

```
Theorem ExistsSomethingTrue : exists P : Prop, P.
```

Bastaría hacer:

```
Proof.
  exists True.
  exact I.
Qed.
```

Notar que luego de ejecutar la táctica `exists True`, Coq nos pide que demostremos `True`. Para ello utilizamos la táctica `exact`. Esta táctica utiliza la prueba que se pasa como argumento para demostrar el subojetivo que debemos probar. En este caso pasamos como argumento `I`, la prueba por definición de `True`.

Universales

La forma de probar universales ya ha sido introducida en ejemplos anteriores cuando utilizamos la táctica `intro`. Pero supongamos que quisieramos demostrar que:

```
Theorem ForAllPObvious : forall P : Prop, P = P.
```

Bastaría hacer:

```
Proof.
  intro P.
  reflexivity.
Qed.
```

La táctica `intro` agrega `P : Prop` a las hipótesis y nos pide demostrar el subobjetivo `P = P`. `reflexivity` resuelve la igualdad `P = P`.

Disyunciones

Para demostrar una disyunción disponemos de las tácticas `left` y `right`. Si quisiéramos probar:

```
Theorem TrueOrFalse : True ∨ False.
```

Bastaría hacer:

```
Proof.
  left.
```

```
  exact I.
Qed.
```

Conjunciones

Para demostrar conjunciones disponemos de la táctica `split` que genera un nuevo subojetivo igual al lado derecho de la conjunción y uno igual al izquierdo. Si quisiéramos probar:

```
Theorem TrueAndTrue : True ∧ True.
```

Bastaría hacer:

```
Proof.
  split.
  - (* left side *)
    exact I.
  - (* right side *)
    exact I.
Qed.
```

Los símbolos −, + y ∗ nos permiten separar los distintos casos. Podemos anidar uno dentro del otro, para hacer la prueba más legible. En este caso como sólo se requiere un nivel, se utiliza únicamente −. Todo lo que se encuentre dentro de `(* *)` es ignorado por Coq como un comentario.

Análisis por casos

Supongamos que quisiéramos demostrar el siguiente teorema:

```
Theorem plusZero : forall n : nat, plus Zero n = n.
```

La demostración es un simple análisis por casos en `n`. La táctica `destruct` realiza esto:

```
Proof.
  intro n.
  destruct n.
  - (* Caso n = Zero *)
    reflexivity.
  - (* Caso n = Succ n' *)
    reflexivity.
Qed.
```

Una vez aplicada la táctica `destruct`, Coq nos informa que debemos demostrar dos subobjetivos. Por un lado debemos demostrar que `plus Zero Zero = Zero` y por el otro que `plus Zero (Succ n') = (Succ n')`.

En el primer caso debemos demostrar `plus Zero Zero = Zero`. Para lograr esto, utilizamos la táctica `reflexivity`. Esta táctica intenta hacer ambos lados de la igualdad idénticos a través de simplicaciones. En este caso, por ejemplo, simplifica el lado izquierdo de la igualdad aplicando la definición de `plus` obteniendo `Zero = Zero`. Como el resultado que obtiene en el lado izquierdo es igual al lado derecho, la táctica tiene éxito.

En el segundo caso debemos demostrar `plus Zero (Succ n') = (Succ n')`. De la misma forma que en el primer caso, `reflexivity` será suficiente para demostrar el caso, puesto que al aplicar la definición de `plus` en el lado izquierdo obtenemos la igualdad `Succ n' = Succ n'`. Nuevamente, como el resultado que obtiene en el lado izquierdo es igual al lado derecho, la táctica tiene éxito.

Inducción Estructural

Para introducir la táctica que permite realizar inducción estructural, tomemos el ejemplo del teorema:

```
Theorem zeroPlus : forall n : nat, plus n Zero = n.
```

Si bien este teorema parece extremadamente similar al anterior, notemos que un análisis con casos no bastará para demostrar `zeroPlus` puesto que la definición de `plus` es recursiva en en el primer argumento, no en el segundo. Para demostrarlo deberemos realizar inducción estructural sobre la variable `n` usando la táctica `induction`:

```
Proof.
  induction n as [| n' IHn].
  + (* Caso n = Zero *)
    reflexivity.
  + (* Caso n = Succ n' *)
    simpl. rewrite IHn. reflexivity.
Qed.
```

En el caso inductivo primero simplificamos el subobjetivo a demostrar utilizando `simpl`, luego aplicamos la hipótesis inductiva utilizando `rewrite IHn` y finalmente demostramos el caso utilizando `reflexivity`. La expresión `as [| n' IHn]` simplemente le da un nombre a `n'` en el caso inductivo y a la hipótesis inductiva.

Otras tácticas

Existen muchas librerías de tácticas disponibles y además el usuario puede definir nuevas tácticas a partir de otras utilizando el lenguaje L_{tac}. A modo de ejemplo, existe una táctica llamada `tauto` que puede demostrar algunas tautologías, una librería que permite resolver problemas sin cuantificadores expresables en la aritmética de Presburger denominado Omega y un conjunto de tácticas para resolver problemas aritméticos en anillos ordenados llamado Micromega.

Para una descripción más detallada puede consultar los manuales de referencia de ltac, omega y micromega que se encuentran en el manual de referencia de Coq https://coq.inria.fr/refman.

4.10 COQ ES UNA LÓGICA INTUICIONISTA

En la semántica de lógica clásica, toda proposición o bien es verdadera o bien es falsa. Esto implica la validez de ciertas proposiciones como $P \vee (\neg P)$ (principio del tercero excluído), $\neg\neg P \to P$ (eliminación de la doble negación) y

$\neg\forall x, \neg P(x) \rightarrow \exists x, P(x)$ (principio de demostración indirecta), para cualquier P. En particular el principio de demostración indirecta nos dice que si sabemos que no es posible que ningún x cumpla P, entonces debe existir un objeto que cumpla P. Si bien este principio parece razonable y es utilizado en matemática frecuentemente como método de demostración, presenta el problema de que permite afirmar la existencia de objetos sin dar un método para construirlos explícitamente.

Las lógicas intuicionistas rechazan estos principios no constructivos a favor de razonamientos que sólo permiten derivaciones contructivas. Para lograr esto, la semántica de las lógicas intuicionistas, en vez de enfocarse exclusivamente en el valor de verdad de una sentencia, da una idea de qué significa que cierto objeto matemático sea una prueba de una proposición. Una proposición será válida en estas semánticas si se puede construir una prueba de la misma. Expresado de otra manera, en vez de responder únicamente a la pregunta de cuándo una proposición es verdadera la semántica de estas lógicas intentan responder a la pregunta de cuándo tenemos una prueba o una evidencia de que una proposición es verdadera.

Teniendo en cuenta esto, resulta evidente que las lógicas intuicionistas no pueden aceptar el principio de tercero excluído: aceptar $P \vee \neg P$ implica aceptar que para cualquier proposición P siempre podemos extraer una prueba de P o una prueba de $\neg P$.

La lógica intuicionista subyacente a Coq es el cálculo de construcciones inductivas. En esta lógica, como en cualquier otra lógica intuicionista, existen ciertas proposiciones clásicas que no pueden ser demostradas. Debido a que a veces podemos llegar a querer formalizar y demostrar estas proposiciones, Coq nos permite asumir el principio de tercero excluído como axioma para poder demostrarlas al costo de perder el contenido computacional de las pruebas.

4.11 BOOL Y PROP

Supongamos que queremos formalizar en Coq una propiedad de los elementos de cierto dominio. Por ejemplo, podríamos querer formalizar en Coq la propiedad "x es par" sobre el dominio de los números naturales, o "x es racional" sobre el dominio de los números reales. Para hacer esto, podemos preguntarnos qué es, formalmente, una propiedad sobre un dominio `T`. Una primera respuesta que podríamos dar es decir que es una función que toma elementos del dominio y devuelve valores de verdad. Vimos que existen dos tipos en Coq para representar valores de verdad: `bool` y `Prop`. Pero entonces ¿cuál es el adecuado? ¿debemos formalizar las propiedades como `P : T → bool` o como `P : T → Prop`? Analicemos cada caso.

Supongamos que queremos formalizar la proposición "x es par" de la definiendo un predicado con codominio `bool`. Esto puede realizarse fácilmente definiendo una función `boolEven` de la siguiente forma:

```
Fixpoint boolEven (n : nat) : bool :=
match n with
| Zero ⇒ true
```

```
| Succ n ⇒ not (boolEven n)
end.
```

Una propiedad interesante de esta formalización es que podemos, para cualquier valor de `T`, computar cuál es su valor de verdad. Por ejemplo, si queremos saber si 42 es par basta con pedirle a Coq que compute la expresión `boolEven 42`. Esta propiedad no se debe a ninguna peculiaridad de nuestro ejemplo. En efecto, cualquier expresión de la forma `P t` con `t : T` puede ser evaluada a alguno de los valores `true` o `false`. El lector podría objetar que es posible definir propiedades que al ser evaluadas, su evaluación no termine. Pero esto no es así puesto que, como ya expresamos anteriormente, en Coq sólo pueden definirse funciones que terminen para cualquier entrada.

Pero entonces, las únicas propiedades formalizables en `bool` son aquellas que son decidibles, es decir, aquellas propiedades que su valor de verdad para cualquier valor del dominio puede ser computado.

Notemos que la formalización de propiedades en `bool` es simplemente la formalización de un procedimiento que determina el valor de verdad de la propiedad para un determinado `t : T`. Es decir que en la definición de la propiedad estamos explicitando la forma en la que para un valor arbitrario del dominio podemos determinar el valor de verdad de la propiedad. En este sentido decimos que `bool` es computacional.

`Prop`, a diferencia de `bool` no es computacional. Es decir, si formalizamos una propiedad como una función `P : T → Prop` y tenemos que `t : T`, no siempre será posible computar con Coq `P t`. De hecho, cuando formalizamos propiedades en `Prop`, no es necesario explicitar un procedimiento que le permita a Coq computar el valor de verdad de `P t`. Aún más, `Prop` nos permite formalizar propiedades que no son decidibles.

Parte II

LA FORMALIZACIÓN

5 LÓGICA MODAL BÁSICA EN COQ

En este capítulo presentamos la formalización de la lógica modal básica en conjunto con la demostración del teorema de invarianza correspondiente. El código fuente está dividido en tres archivos: `Syntax.v`, `Semantics.v` y `InvarianceTheorem.v`. En `Syntax.v` formalizamos las nociones sintácticas descriptas en la Sección 5.1. En `Semantics.v` damos las formalizaciones de las nociones de satisfacibilidad, bisimulación y equivalencia modal. Cubrimos esto en las Secciones 5.2, 5.3 y 5.4. Finalmente `InvarianceTheorem.v` contiene la demostración del teorema de invarianza, que daremos en la Sección 5.5.

5.1 EL LENGUAJE

Primero debemos formalizar la Definición 2.1 del lenguaje de la lógica modal básica. Esto implica formalizar el conjunto `PROP` de los símbolos proposicionales y el conjunto de fórmulas modales $\mathcal{ML}$. Para lo primero definimos el tipo inductivo:

```
Inductive prop : Set :=
p : nat → prop.
```

A partir de esto, podemos definir el conjunto de fórmulas modales $\mathcal{ML}$:

```
Inductive form : Set :=
| Atom   : prop → form
| Bottom : form
| If     : form → form → form
| Diam   : form → form.
```

Interpretamos esta definición de la siguiente manera:

- Dadas una proposición $p \in$ `PROP` y su correspondiente formalización `p : prop`, `Atom p` es una formalización de la fórmula atómica p.
- `Bottom` formaliza $\bot$
- Dadas dos fórmulas $\varphi, \psi \in \mathcal{ML}$ y sus respectivas formalizaciones `phi : form` y `psi : form`, `If phi psi` es una fórmula válida que formaliza la fórmula $\varphi \to \psi$.
- Dada una fórmula $\varphi \in \mathcal{ML}$ y su respectiva formalización `phi`, `Diam phi` formaliza la fórmula $\Diamond\varphi$.

Para la formalización de la Notación 2.1 en Coq, utilizamos el comando `Definition`:

```
Definition Not (phi : form) : form := If phi Bottom.

Definition Top : form := Not Bottom.
```

```
Definition And (phi psi : form) : form := Not (If phi (Not psi)).

Definition Or (phi psi : form) : form := If (Not phi) psi.

Definition Iif (phi psi : form) : form := And (If phi psi) (If psi phi).

Definition Box (phi : form) : form := Not (Diam (Not phi)).
```

Finalmente incorporamos la siguiente notación para que escribir fórmulas sea menos tedioso:

```
Notation "p ∧' q" := (And p q) (at level 80, right associativity).

Notation "p ∨' q" := (Or p q) (at level 85, right associativity).

Notation "~' p" := (Not p) (at level 70, right associativity).

Notation "p →' q" := (If p q) (at level 90, right associativity).

Notation "p ↔' q" := (Iif p q) (at level 95, right associativity).

Notation "[] F" := (Box F) (at level 65, right associativity).

Notation "<> F" := (Diam F) (at level 65, right associativity).
```

Estos comandos, además de crear la nueva notación establecen la asociatividad de los operadores. En particular, declaramos a todos estos operadores como asociativos a la derecha. Por otra parte, también estamos estableciendo los niveles de precedencia de los mismos. Un nivel de precedencia bajo une más que un nivel alto. Por ejemplo, dado que `∨'` tiene un nivel de precedencia de 85 mientras que `~'` tiene un nivel de precedencia de 70, la expresión `~' p ∨' p` se puede leer, gracias a esta notación, como `(~' p) ∨' p`.

5.2 MODELOS DE KRIPKE

En esta sección formalizamos la noción de modelo de Kripke dada en la Definición 2.2. En nuestra formalización evitamos utilizar *Record Types* para definir el modelo de Kripke. Hicimos esto simplemente porque las demostraciones resultaban menos legibles cuando lo utilizamos. Es decir que el modelo de Kripke no está formalizado como un tipo, sino como un trío de valores cuyos tipos dependen unos de los otros. El primero es `W`, el dominio. El segundo es la relación de accesibilidad `R` y el último es la función de valuación `V`. Los tipos de cada uno son los siguientes:

```
W : Set
R : W → W → Prop
V : W → prop → Prop
```

Notar la dependencia de los tipos `R` y `V` en el conjunto `W`, que es un valor.

El tipo de `V` merece una explicación. En la Definición 2.2 que dimos de modelo de Kripke, definimos a la función de valuación como una función de la forma $V :$

PROP $\to 2^W$ donde W era el dominio del modelo. Dados un símbolo proposicional p $\in$ PROP, un punto $w \in W$ y sus respectivas formalizaciones p : prop, w : W, formalizaremos $w \in V(p)$ como V w p.

5.3 SATISFACIBILIDAD

La Definición 2.3 de satisfacibilidad fue formalizada utilizando una función recursiva sobre la fórmula:

```
Fixpoint satisfies
        (W : Set)
        (R : W → W → Prop)
        (V : W → prop → Prop)
        (w : W)
        (phi : form) : Prop :=
match phi with
| Atom v ⇒ (V w v)
| Bottom ⇒ False
| If phi psi ⇒ satisfies W R V w phi → satisfies W R V w psi
| Diam phi ⇒ exists v : W, R w v ∧ satisfies W R V v phi
end.
```

También se definió una notación para hacer el código más legible:

```
Notation "# W , R , L >> w |= phi" := (satisfies W R L w phi) (at level 30).
```

5.4 BISIMULACIONES

En esta sección damos la formalización de la noción de bisimulación presentada en la Definición 2.6 y del concepto de equivalencia modal presentado en la Definición 2.4. Estas formalizaciones constituirán el último paso que nos permitirá demostrar formalmente el teorema de invarianza.

La equivalencia modal fue formalizada de la siguiente manera:

```
Definition equivalent_at_points W R L W' R' L' w w' :=
  forall (phi : form), (# W , R , L >> w |= phi) ↔ (# W' , R' , L' >> w' |= phi).
```

Notemos que esta formalización es una traducción directa de la Definición 2.4.

Para formalizar la noción de bisimulación primero definimos cada una de las tres condiciones por separado y luego utilizamos tales formalizaciones como funciones para dar la definición de bisimulación:

```
Definition atomic_harmony
          (W : Set) (L : W → prop → Prop)
          (W' : Set) (L' : W' → prop → Prop)
          (Z : W → W' → Prop) : Prop :=
  forall w w', Z w w' → L w = L' w'.

Definition zig
          (W : Set) (R : W → W → Prop)
          (W' : Set) (R' : W' → W' → Prop)
```

```
          (Z : W → W' → Prop) : Prop :=
  forall w w' v, Z w w' → R w v → (exists v' : W', Z v v' ∧ R' w' v').

Definition zag
          (W : Set) (R : W → W → Prop)
          (W' : Set) (R' : W' → W' → Prop)
          (Z : W → W' → Prop) : Prop :=
  forall w w' v', Z w w' → R' w' v' → (exists v : W, Z v v' ∧ R w v).
```

Luego definimos una función que formaliza la pregunta ¿es esta función entre modelos una bisimulación entre ellos?:

```
Definition bisimulation W R L W' R' L' Z :=
  (atomic_harmony W L W' L' Z) ∧ (zig W R W' R' Z) ∧ (zag W R W' R' Z).
```

Después definimos una función que formaliza la pregunta ¿es esta relación entre elementos de modelos una bisimulación entre ellos y además cumple que relaciona el punto `w` con el punto `w'`?:

```
Definition bisimulation_at_points W R L W' R' L' Z w w' :=
  bisimulation W R L W' R' L' Z ∧ Z w w'.
```

Y a continuación definimos la noción de bisimilaridad, es decir, que exista alguna relación que sea una bisimulación:

```
Definition bisimulable W R L W' R' L' :=
  exists Z, bisimulation W R L W' R' L' Z.

Definition bisimulable_at_points W R L W' R' L' w w':=
  exists Z, bisimulation W R L W' R' L' Z ∧ Z w w'.
```

5.5 TEOREMA DE INVARIANZA

Primero debemos enunciar el Teorema 2.1 en esta formalización:

```
Theorem InvarianceUnderBisimulation :
  forall W R L W' R' L' w w',
    bisimulable_at_points W R L W' R' L' w w' →
    equivalent_at_points W R L W' R' L' w w'.
```

Ahora empecemos la demostración con el comando `Proof.`.

Lo primero que vamos a hacer en la demostración es eliminar los cuantificadores universales mediante la táctica `intros`:

```
intros W R L W' R' L' w w'.
```

Ahora sabemos que:

```
W : Set
R : W → W → Prop
L : W → prop → Prop
W' : Set
R' : W' → W' → Prop
L' : W' → prop → Prop
w : W
w' : W'
```

y debemos demostrar que:

```
bisimulable_at_points W R L W' R' L' w w' →
equivalent_at_points W R L W' R' L' w w'
```

Ahora sustituyamos `bisimulable_at_points` y `equivalent_at_points` por sus definiciones usando la táctica `unfold`:

```
unfold bisimulable_at_points.
unfold equivalent_at_points.
```

Recordemos que `bisimulable_at_points` está definido en función de `bisimulation`, por lo que será necesario usar `unfold` una vez más:

```
unfold bisimulation.
```

Coq nos responde que debemos demostrar:

```
(exists Z : W → W' → Prop,
   (atomic_harmony W L W' L' Z ∧ zig W R W' R' Z ∧ zag W R W' R' Z) ∧
   Z w w') →
forall phi : form, # W, R, L >> w |= phi ↔ # W', R', L' >> w' |= phi
```

Ordenemos un poco esto. Primero asumiremos el antecedente, luego tomaremos un ejemplo de su cuantificador existencial mediante la táctica `destruct` y finalmente separaremos las condiciones de armonía atómica, zig, zag y el hecho de que `Z` relaciona `w` con `w'` en cuatro hipótesis separadas también mediante la táctica `destruct`:

```
intro H.
destruct H as [Z H].
destruct H as [H HZww'].
destruct H as [HAtomicHarmony H].
destruct H as [HZig HZag].
```

Esto nos agrega las siguientes hipótesis:

```
Z : W → W' → Prop
HAtomicHarmony : atomic_harmony W L W' L' Z
HZig : zig W R W' R' Z
HZag : zag W R W' R' Z
HZww' : Z w w'
```

y deja el subobjetivo a demostrar de la siguiente manera:

```
forall phi : form, # W, R, L >> w |= phi ↔ # W', R', L' >> w' |= phi
```

Eliminamos el cuantificador universal del subobjetivo introduciendo `phi`:

```
intro phi.
```

por lo que el subobjetivo queda:

```
# W, R, L >> w |= phi ↔ # W', R', L' >> w' |= phi
```

Ya casi estamos listos para utilizar inducción estructural para demostrar el teorema. Sin embargo, debemos fortalecer la hipótesis inductiva. Cuando demostramos el Teorema 2.1 informalmente, vimos que la proposición a demostrar por inducción en las fórmulas era:

Para todo $w \in W$ y para todo $w' \in W'$, si wZw' entonces $\mathcal{M}, w \equiv_{\mathcal{ML}} \mathcal{M}', w'$.

Notar la cuantificación universal de w y de w'. Si ahora intentaramos probar el subobjetivo por inducción estructural sobre `phi`, no podríamos puesto que la hipótesis inductiva no sería lo suficientemente fuerte. Es por esto que debemos generalizar el objetivo a demostrar para cualquier `w` y cualquier `w'`. Para esto, Coq provee una táctica que se encarga de esto:

```
generalize dependent w'.
generalize dependent w.
```

El subobjetivo queda idéntico a la hipótesis inductiva de la demostración informal:

```
forall (w : W) (w' : W'),
Z w w' → # W, R, L >> w |= phi ↔ # W', R', L' >> w' |= phi
```

y ahora ya no disponemos de las hipótesis `w : W` y `w' : W'`.

Una vez realizado el fortalecimiento de la hipótesis inductiva podemos aplicar inducción estructural en `phi`:

```
induction phi as [p | | phi IHphi psi IHpsi | psi IH].
```

Notemos que la sintaxis `as [p | | phi IHphi psi IHpsi | psi IH]` le indica a Coq cómo debe nombrar las distintas hipótesis de los distintos casos. Más específicamente:

- `p` será el nombre del símbolo proposicional del caso base `Atom p`.
- `phi psi` serán las fórmulas de menor tamaño del caso inductivo `phi → psi`, mientras que `IHphi` y `IHpsi` serán los nombres de sus respectivas hipótesis inductivas.
- `phi` será la fórmula de menor tamaño del caso inductivo `Diamond phi`, mientras que `IH` será el nombre de su hipótesis inductiva.

Ahora continuemos con cada caso.

```
▷ phi = Atom p
```

Debemos demostrar que:

```
forall (w : W) (w' : W'),
Z w w' → # W, R, L >> w |= Atom p ↔ # W', R', L' >> w' |= Atom p
```

La demostración es sencilla. Primero aplicamos la definición de `satisfies`, luego introducimos las variables `w` y `w'` y la hipótesis `Z w w'` que llamaremos `HZww'`:

```
unfold satisfies. intros w w' HZww'.
```

En este punto nos queda demostrar que:

```
L w p ↔ L' w' p
```

Pero sabemos que por armonía atómica esto debe ser cierto:

```
rewrite (HAtomicHarmony w w' HZww').
tauto.
```

```
▷ phi = Bottom
```

Debemos demostrar que:

```
forall (w : W) (w' : W'),
Z w w' → # W, R, L >> w |= Bottom ↔ # W', R', L' >> w' |= Bottom
```

En este caso basta hacer:

```
tauto.
```

```
▷ phi = If phi psi
```

Debemos demostrar que:

```
forall (w : W) (w' : W'),
Z w w' →
# W, R, L >> w |= (phi → ' psi) ↔
# W', R', L' >> w' |= (phi → ' psi)
```

Primero aplicamos la definición de `satisfies` pero teniendo el cuidado de no aplicarla muchas veces. Para esto hacemos:

```
unfold satisfies. fold satisfies.
```

Luego introducimos las variables `w` y `w'` y la hipótesis `Z w w'` que llamaremos `HZww'`:

```
intros w w' HZww'.
```

En este punto debemos demostrar dos implicaciones:

```
(# W, R, L >> w |= phi → # W, R, L >> w |= psi) ↔
(# W', R', L' >> w' |= phi → # W', R', L' >> w' |= psi)
```

Por lo que usaremos la táctica `split` para realizar un análisis por casos.

Es importante notar en este punto que las demostraciones de las dos implicaciones son idénticas. Veremos únicamente la implicación de izquierda a derecha y el lector podrá confirmar fácilmente que la otra es idéntica.

Mediante la siguiente táctica:

```
intros H Hsat.
```

introducimos las siguientes hipótesis:

```
H: # W, R, L >> w |= phi → # W, R, L >> w |= psi
Hsat: # W', R', L' >> w' |= phi
```

Nos queda demostrar:

```
# W', R', L' >> w' |= psi
```

Usaremos la hipótesis inductiva de `psi` que nos dice que para demostrar `# W', R', L' >> w' |= psi` basta probar `# W, R, L >> w |= psi`. Luego aplicaremos la hipótesis `H` que reducirá el objetivo a demostrar `# W, R, L >> w |= phi`. Pero la hipótesis inductiva de `phi` afirma que para demostrar esto es suficiente demostrar `# W', R', L' >> w' |= phi`, algo que sabemos que es cierto por `Hsat`. Teniendo en cuenta lo anterior, la prueba quedaría:

```
apply (IHpsi w w' HZww').
apply H.
apply (IHphi w w' HZww').
apply Hsat.
```

Como la implicación de izquierda a derecha y la implicación de derecha a izquierda son idénticas podemos escribir la demostración de la siguiente manera:

```
unfold satisfies. fold satisfies. intros w w' HZww'.
split;
  intros H Hsat;
  apply (IHpsi w w' HZww');
  apply H;
  apply (IHphi w w' HZww');
  apply Hsat.
```

De esta manera se aplican todas las tácticas a ambos subobjetivos (es decir, a ambas implicaciones).

```
▷ phi = Diamond phi
```

En este caso debemos demostrar que:

```
forall (w : W) (w' : W'),
Z w w' →
# W, R, L >> w |= (<> phi) ↔ # W', R', L' >> w' |= (<> phi)
```

Análogamente al caso anterior empezamos con las siguientes tácticas:

```
unfold satisfies. fold satisfies. intros w w' HZww'.
```

El objetivo a demostrar nos queda:

```
(exists v : W, R w v ∧ # W, R, L >> v |= phi) ↔
(exists v : W', R' w' v ∧ # W', R', L' >> v |= phi)
```

Como son dos casos, los analizamos por separado con la táctica `split`. Debido a que ambos casos son completamente análogos aquí sólo presentaremos la demostración de uno de ellos, la demostración completa puede verse en el Capítulo C del Apéndice.

Demostraremos el implica de izquierda a derecha. Mediante la táctica:

```
intro H.
```

introducimos el antecedente del implica con el nombre `H`:

```
H: exists v : W, R w v ∧ # W, R, L >> v |= phi
```

A continuación tomaremos un ejemplo de este existencial que denominaremos `v`. Por `H`, sabemos que `v` tiene tipo `W` y que satisface dos propiedades que nombraremos `HRwv` y `Hsatv`:

```
HRwv: R w v
Hsatv: # W, R, L >> v |= phi
```

Para lograr esto basta hacer:

```
destruct H as [v [HRwv Hsatv]].
```

Como nos interesa probar que:

```
exists v' : W', R' w' v' ∧ # W', R', L' >> v' |= phi
```

vamos a aplicar la propiedad zig:

```
HZig : forall (w : W) (w' : W') (v : W),
       Z w w' → R w v → exists v' : W', Z v v' ∧ R' w' v'
```

con argumentos `w`, `w'`, `v` y `HZww'` en `HRwv` para obtener:

```
exists v' : W', Z v v' ∧ R' w' v'
```

Notemos que hacer esto nos permitirá dar un ejemplo de un `v'` que cumple `R' w' v'`. Una vez que hagamos esto, tendremos que demostrar que ese mismo `v'` cumple `# W', R', L' >> v' |= phi`. Para esto último utilizaremos la hipótesis inductiva y el hecho de que `Z v v'`.

Como dijimos, primero tomamos un ejemplo del cuantificador existencial que llamaremos `v'` y nombraremos a las dos propiedades que por `Hzig` sabemos que posee `v'` de la siguiente manera:

```
HZvv' : Z v v'
HR'w'v' : R' w' v'
```

Todo esto lo podemos lograr haciendo:

```
unfold zig in HZig.
apply (HZig w w' v HZww') in HRwv as [v' [HZvv' HR'w'v']].
```

Ahora tomamos el `v'` como nuestro candidato para demostrar el existencial:

```
exists v'.
```

por lo que sólo nos queda probar:

```
R' w' v' ∧ # W', R', L' >> v' |= phi
```

Para esto, separamos en dos casos usando la siguiente táctica:

```
split.
```

Notemos que ya sabemos que `R' w' v'` por la hipótesis `HR'w'v'`, por lo que basta con usar la táctica `assumption`. para este caso. Para el otro caso, basta aplicar la hipótesis inductiva `IH` que afirma:

```
IH : forall (w : W) (w' : W'),
     Z w w' →
     # W, R, L >> w |= phi ↔ # W', R', L' >> w' |= phi
```

con los argumentos `v` `v'` y `HZvv'` para reducir el subobjetivo a `# W', R', L' >> v' |= phi`, algo que podemos que demostrar usando `assumption`. puesto que `Hsatv' : # W', R', L' >> v' |= phi` es una de nuestras hipótesis.

Por lo tanto, este análisis por casos quedaría:

```
* assumption.
* apply (IH v v' HZvv'). assumption.
```

6 LÓGICA MODAL DINÁMICA EN COQ

En este capítulo presentamos la formalización de la lógica modal dinámica en conjunto con la demostración del teorema de invarianza correspondiente. El código fuente está dividido en tres archivos: `Syntax.v`, `Semantics.v` y `InvarianceTheorem.v`. En `Syntax.v` formalizamos las nociones sintácticas descriptas en la Sección 6.1. En `Semantics.v` damos las formalizaciones de las nociones de satisfacibilidad, bisimulación y equivalencia modal. Cubrimos esto en las Secciones 6.2, 6.3 y 6.4. Finalmente `InvarianceTheorem.v` contiene la demostración del teorema de invarianza, que daremos en la Sección 6.5.

6.1 EL LENGUAJE

Lo primero que haremos será definir el conjunto `PROP` de los símbolos proposicionales y el conjunto de fórmulas modales $\mathcal{ML}$ dados en la Definición 3.1. Al igual que en el Capítulo 5, definimos:

```
Inductive prop : Set :=
p : nat → prop.
```

Antes de continuar declararemos el conjunto de los operadores dinámicos como una variable de Coq:

```
Variable Dyn : Set.
```

Este conjunto es necesario para formalizar un lenguaje dinámico por cada conjunto de operadores dinámicos que elijamos. Notemos que esto tiene que ver con que en la Definición 3.1 cuando definimos $\mathcal{ML}$ se lo hizo en función de un conjunto de operadores dinámicos `DYN`.

A partir de esto, podemos definir el conjunto de fórmulas modales $\mathcal{ML}$:

```
Inductive form : Type :=
| Atom    : prop → form
| Bottom  : form
| If      : form → form → form
| Diam    : form → form
| DynDiam : Dyn → form → form.
```

Interpretamos esta definición de la siguiente manera:

- Dadas una proposición `p` $\in$ `PROP` y su correspondiente formalización `p : prop`, `Atom p` es una formalización de la fórmula atómica `p`.
- `Bottom` formaliza $\bot$
- Dadas dos fórmulas $\varphi, \psi \in \mathcal{ML}$ y sus respectivas formalizaciones `phi : form` y `psi : form`, `If phi psi` es una fórmula válida que formaliza la fórmula $\varphi \to \psi$.

- Dada una fórmula $\varphi \in \mathcal{ML}$ y su respectiva formalización `phi`, `Diam phi` formaliza la fórmula $\Diamond\varphi$.
- Dados una fórmula $\varphi \in \mathcal{ML}$ y un operador dinámico $\blacklozenge_d$ y su respectivas formalizaciones `phi` y `d`, `DynDiam d phi` formaliza la fórmula $\blacklozenge_d\varphi$.

Análogamente a como hicimos en el cápitulo anterior, formalizamos las abreviaciones dadas en la Notación 3.1 usando el comando `Definition`:

```
Definition Not (phi : form) : form := If phi Bottom.

Definition Top : form := Not Bottom.

Definition And (phi psi : form) : form := Not (If phi (Not psi)).

Definition Or (phi psi : form) : form := If (Not phi) psi.

Definition Iif (phi psi : form) : form := And (If phi psi) (If psi phi).

Definition Box (phi : form) : form := Not (Diam (Not phi)).

Definition DynBox (d : Dyn) (phi : form) : form := Not (DynDiam d (Not phi)).
```

Finalmente incorporamos la siguiente notación para que escribir fórmulas sea menos tedioso:

```
Notation "p ∧' q" := (And p q) (at level 80, right associativity).

Notation "p ∨' q" := (Or p q) (at level 85, right associativity).

Notation "~' p" := (Not p) (at level 70, right associativity).

Notation "p →' q" := (If p q) (at level 90, right associativity).

Notation "p ↔' q" := (Iif p q) (at level 95, right associativity).

Notation "[m] phi" := (Box phi) (at level 65, right associativity).

Notation "<m> phi" := (Diam phi) (at level 65, right associativity).

Notation "<o> d phi" := (DynDiam d phi) (at level 65, right associativity).

Notation "[o] d phi" := (DynBox d phi) (at level 65, right associativity).
```

6.2 MODELOS DE KRIPKE Y FUNCIONES DE ACTUALIZACIÓN DE MODELOS

Los modelos de Kripke fueron formalizados igual que como se explica en la Sección 5.2.

Recordemos que una función de actualización de modelos (ver la Definición 3.2) para un dominio W es una función $f_W : W \times 2^{W^2} \to 2^{W \times 2^{W^2}}$ que toma un estado en W y una relación binaria sobre W y devuelve un conjunto de

posibles actualizaciones al estado de evaluación y la relación de accesibilidad. Para entender la formalización de estas funciones es mejor ver algunos ejemplos más sencillos.

Ejemplo 6.1. Sean A un conjunto y `A` su formalización en Coq como tipo. ¿Cómo formalizamos el conjunto de partes 2^A de A? Podemos pensar a un elemento del conjunto de partes de A (es decir, a un subconjunto de A) como una función que dado un elemento de A determina si ese elemento pertenece o no a ese conjunto. Es decir que 2^A puede formalizarse con el tipo `A → Prop`.

Ejemplo 6.2. Sea A un conjunto y `A` su formalización en Coq como tipo. ¿Cómo formalizamos el conjunto de las relaciones binarias sobre A? Notemos lo siguiente: una relación binaria R sobre A puede pensarse como una función que dados dos elementos $a, b \in A$ nos devuelva un valor $Verdadero$ si ocurre que aRb y un valor $Falso$ si no ocurre que aRb. El lector podría pensar que entonces es una buena idea formalizar el conjunto de las relaciones binarias de la siguiente manera: `A → A → Bool`. Pero esto sería incorrecto, puesto que implicaría que siempre podemos decidir si aRb o no, pero no queremos limitarnos a los casos en donde podemos efectivamente decidir esto, queremos formalizar cualquier tipo de relaciones. Por esto, la formalización adecuada es `A → A → Prop`.

Ahora podemos formalizar el conjunto de las funciones de actualización de modelos: primero notemos que el tipo debe ser un tipo dependiente en el conjunto W. Por lo tanto, si `W` es la formalización de W, el tipo que buscamos tendrá la forma `forall (W : Set), T` donde `T` dependerá de `W`. Además sería útil representar con `R` al tipo `W → W → Prop` de las relaciones binarias sobre `W`. Hasta ahora tenemos que el tipo buscado tendrá la forma `forall (W : Set), let R := (W → W → Prop) in T'` donde `T'` depende de `W` y de `R`. Finalmente notemos que una función de actualización de modelos puede pensarse como una función que toma un punto en W, una relación binaria sobre W y retorna un conjunto de pares de puntos y relaciones binarias sobre W. Pero en vez de este conjunto podríamos retornar una función que dado un punto en W y una relación binaria sobre W determinara si el punto y la relación pertenecen a ese conjunto. Nuestra formalización queda:

```
Definition model_update_function : Type := forall (W : Set),
  let R := (W → W → Prop) in W → R → (W → R → Prop).
```

Como el nombre es muy largo y muy usado, definimos la siguiente abreviación:

```
Definition muf := model_update_function.
```

Notemos que en esta formalización una función de actualización de modelos depende de un conjunto `W : Set`. Por lo tanto podemos pensar el tipo `muf` como el tipo que formaliza el tipo de las familias de funciones de actualización de modelos de la clase de todos los modelos. Siendo más preciso, un habitante del tipo `muf` formaliza una función que dado un dominio cualquiera devuelve una función de actualización de modelos sobre ese dominio.

6.3 SATISFACIBILIDAD

Antes de dar la formalización de satisfacibilidad, debemos introducir una variable que llamaremos `F`:

```
Definition Dyn := Syntax.Dyn.
Variable F : Dyn → muf.
```

En la primera línea, para mejorar la legibilidad del código definimos `Dyn` como `Syntax.Dyn`. Luego damos la definición de esta variable. Notemos que se trata de una función que por cada operador dinámico nos asigna una función de actualización de modelos. Cada vez que se quiera trabajar con una nuevo conjunto de operadores dinámicos se deberá definir tanto un conjunto `Dyn` de operadores dinámicos y una función `F` que a cada operador dinámico le asigne un comportamiento, es decir una función de actualización de modelos.

La definición de satisfacibilidad fue formalizada utilizando una función recursiva sobre la fórmula:

```
Fixpoint satisfies
        (W : Set)
        (R : W → W → Prop)
        (L : W → prop → Prop)
        (w : W)
        (phi : form) : Prop :=
match phi with
| Atom v ⇒ (L w v)
| Bottom ⇒ False
| phi1 →' phi2 ⇒ (satisfies W R L w phi1) → (satisfies W R L w phi2)
| Diam phi ⇒
   exists v : W, R w v ∧ (satisfies W R L v phi)
| DynDiam d phi ⇒
  let fw := F d W in
  exists (v : W) (R' : W → W → Prop), fw w R v R' ∧
                          satisfies W R' L v phi
end.
```

También se definió una notación para hacer el código más legible:

```
Notation "# W , R , L >> w |= phi" := (satisfies W R L w phi) (at level 30).
```

6.4 BISIMULACIONES

En esta sección damos la formalización de la noción de bisimulación presentada en la Definición 3.7 y del concepto de equivalencia modal presentado en la Definición 3.6. Estas formalizaciones constituirán el último paso que nos permitirá demostrar formalmente el teorema de invarianza..

La equivalencia modal fue formalizada de la misma manera que en la Sección 5.4:

```
Definition equivalent_at_points W R L W' R' L' w w' :=
  forall (phi:form), (# W , R , L >> w |= phi) ↔
              (# W' , R' , L' >> w' |= phi).
```

Antes de empezar a formalizar la noción de bisimulación para estas lógicas, definamos el tipo de las relaciones entre modelos. A diferencia de las lógicas modales básicas, el tipo de las relaciones entre modelos que requerimos para definir las bisimulaciones no sólo relaciona puntos de los dos modelos, sino que relaciona pares que en la primer componente tienen un punto y en la segunda una relación binaria sobre el dominio del modelo. Dadas dos formalizaciones de dominios `W` y `W'` formalizamos la relación entre estos dominios como:

```
Definition model_to_model_relation W W' : Type :=
  W → (W → W → Prop) → W' → (W' → W' → Prop) → Prop.
```

Análogo a como hicimos en el capítulo anterior, para formalizar la noción de bisimulación primero definimos cada una de las cinco condiciones por separado y luego utilizamos estas condiciones formalizadas como funciones para dar la definición de bisimulación:

```
Definition atomic_harmony
           (W : Set) (L : W → prop → Prop)
           (W' : Set) (L' : W' → prop → Prop)
           (Z : model_to_model_relation W W') : Prop :=
  forall w S w' S',
  Z w S w' S' → L w = L' w'.

Definition zig
           (W : Set)
           (W' : Set)
           (Z : model_to_model_relation W W') : Prop :=
  forall w S w' S' v, Z w S w' S' →
    S w v → (exists v' : W', Z v S v' S' ∧ S' w' v').

Definition zag
           (W : Set)
           (W' : Set)
           (Z : model_to_model_relation W W') : Prop :=
  forall w S w' S' v', Z w S w' S' →
    S' w' v' → (exists v : W, Z v S v' S' ∧ S w v).

Definition f_zig
           (W : Set)
           (W' : Set)
           (Z : model_to_model_relation W W')
           (f : muf) : Prop :=
  let (fw, fw') := (f W, f W') in
  forall w S w' S' v T, Z w S w' S' →
    fw w S v T → (exists (v' : W') T', fw' w' S' v' T' ∧ Z v T v' T').

Definition f_zag
           (W : Set)
           (W' : Set)
           (Z : model_to_model_relation W W')
           (f : muf) : Prop :=
  let (fw, fw') := (f W, f W') in
  forall w S w' S' v' T', Z w S w' S' →
    fw' w' S' v' T' → (exists (v : W) T, fw w S v T ∧ Z v T v' T').
```

Análogo a como hicimos en el capítulo anterior damos las siguientes definiciones:

```
Definition bisimulation W L W' L' Z :=
  (atomic_harmony W L W' L' Z) /\ (zig W W' Z) /\ (zag W W' Z) /\
  (forall d : Dyn, (f_zig W W' Z (F d))) /\
  forall d : Dyn, (f_zag W W' Z (F d)).

Definition bisimulation_at_points W R L W' R' L' Z w w' :=
  bisimulation W L W' L' Z /\ Z w R w' R'.

Definition bisimulable W L W' L' :=
  exists Z, bisimulation W L W' L' Z.

Definition bisimulable_at_points W R L W' R' L' w w':=
  exists Z, bisimulation W L W' L' Z /\ Z w R w' R'.
```

6.5 TEOREMA DE INVARIANZA

Primero debemos enunciar el Teorema 3.1 en esta formalización:

```
Theorem InvarianceUnderBisimulation :
  forall W R L W' R' L' w w',
    bisimulable_at_points W R L W' R' L' w w' →
    equivalent_at_points W R L W' R' L' w w'.
```

Empezamos la demostración con el comando `Proof.`.

Lo primero que vamos a hacer en la demostración es eliminar los cuantificadores universales mediante la táctica `intros`:

```
intros W R L W' R' L' w w'.
```

Ahora sabemos que:

```
W : Set
R : W → W → Prop
L : W → prop → Prop
W' : Set
R' : W' → W' → Prop
L' : W' → prop → Prop
w : W
w' : W'
```

y debemos demostrar que:

```
bisimulable_at_points W R L W' R' L' w w' →
equivalent_at_points W R L W' R' L' w w'
```

Ahora sustituyamos `bisimulable_at_points` y `equivalent_at_points` por sus definiciones usando la táctica `unfold`:

```
unfold bisimulable_at_points.
unfold equivalent_at_points.
```

Recordemos que `bisimulable_at_points` está definido en función de `bisimulation`, por lo que será necesario usar `unfold` una vez más:

```
unfold bisimulation.
```

Coq nos responde que debemos demostrar:

```
(exists Z : model_to_model_relation W W',
   (atomic_harmony W L W' L' Z ∧
    zig W W' Z ∧
    zag W W' Z ∧
    (forall d : Dyn, f_zig W W' Z (Semantics.F d)) ∧
    (forall d : Dyn, f_zag W W' Z (Semantics.F d))) ∧
   Z w R w' R') →
forall phi : form, # W, R, L >> w |= phi ↔ # W', R', L' >> w' |= phi
```

Recordemos que habíamos asumido en `Semantics.v` que existía una función `Semantics.F : Dyn → muf` que para cada operador dinámico `d : Dyn` le asignaba una función de actualización de modelos.

Ordenaremos un poco esto creando nuevas hipótesis con nombres adecuados. Primero asumiremos el antecedente, luego tomaremos un ejemplo de su cuantificador existencial mediante la táctica `destruct` y finalmente separaremos las condiciones de armonía atómica, zig, zag, fzig, fzag y el hecho de que `Z` relaciona `w` y `R` con `w'` y `R'` en seis hipótesis separadas también mediante la táctica `destruct`:

```
intro H.
destruct H as [Z H].
destruct H as [H HZwRw'R'].
destruct H as [HAtomicHarmony H].
destruct H as [HZig H].
destruct H as [HZag H].
destruct H as [HFZig HFZag].
```

Esto nos agrega las siguientes hipótesis:

```
HAtomicHarmony : atomic_harmony W L W' L' Z
HZig : zig W W' Z
HZag : zag W W' Z
HFZig : forall d : Dyn, f_zig W W' Z (Semantics.F d)
HFZag : forall d : Dyn, f_zag W W' Z (Semantics.F d)
HZwRw'R' : Z w R w' R'
```

y deja el subobjetivo a demostrar de la siguiente manera:

```
forall phi : form, # W, R, L >> w |= phi ↔ # W', R', L' >> w' |= phi
```

Eliminamos el cuantificador universal del subobjetivo introduciendo `phi`:

```
intro phi.
```

por lo que el subobjetivo queda:

```
# W, R, L >> w |= phi ↔ # W', R', L' >> w' |= phi
```

Ya casi estamos listos para utilizar inducción estructural para demostrar el teorema. Sin embargo, debemos fortalecer la hipótesis inductiva. Cuando demostramos el Teorema 3.1 informalmente, vimos que la proposición a demostrar por inducción en las fórmulas era:

$$\begin{gathered}\text{Para todo } w \in W, w' \in W', R \subseteq W \times W, R' \subseteq W' \times W', \\ \text{y para todo } Z \in (W \times 2^{W^2}) \times (W' \times 2^{W'^2}), \\ \text{si } Z : \mathcal{M}_R, w \leftrightarrows_{\mathcal{ML}(\mathtt{DYN})} \mathcal{M}'_{R'}, w' \text{ entonces } \mathcal{M}_R, w \equiv_{\mathcal{ML}(\mathtt{DYN})} \mathcal{M}'_{R'}, w'.\end{gathered}$$

Notar la cuantificación universal de w, w', R, R′ y Z. Si ahora intentaramos probar el subobjetivo por inducción estructural sobre `phi`, no podríamos puesto que la hipótesis inductiva no sería lo suficientemente fuerte. Es por esto que debemos generalizar el objetivo a demostrar para cualquier `w`, `w'`, `R`, `R'` y cualquier `Z`. Para esto, Coq provee una táctica que se encarga de esto:

```
generalize dependent Z.
generalize dependent R'.
generalize dependent R.
generalize dependent w'.
generalize dependent w.
```

El subobjetivo queda equivalente a la hipótesis inductiva de la demostración informal:

```
forall (w : W) (w' : W')
  (R : W → W → Prop) (R' : W' → W' → Prop)
  (Z : model_to_model_relation W W'),
atomic_harmony W L W' L' Z →
zig W W' Z →
zag W W' Z →
(forall d : Dyn, f_zig W W' Z (Semantics.F d)) →
(forall d : Dyn, f_zag W W' Z (Semantics.F d)) →
Z w R w' R' → # W, R, L >> w |= phi ↔ # W', R', L' >> w' |= phi
```

y ahora ya no disponemos de las hipótesis `w : W` y `w' : W'`, `R : W → W → Prop`, `R' : W' → W' → Prop` y `Z : model_to_model_relation W W'`.

Una vez realizado el fortalecimiento de la hipótesis inductiva podemos aplicar inducción estructural en `phi`:

```
induction phi as [p | | phi IHphi psi IHpsi | phi IH | d phi IH].
```

Notemos que la sintaxis `as [p | | phi IHphi psi IHpsi | psi IH]` le indica a Coq cómo debe nombrar las distintas hipótesis de los distintos casos. Más específicamente:

- `p` será el nombre del símbolo proposicional del caso base `Atom p`.
- `phi psi` serán las fórmulas de menor tamaño del caso inductivo `phi → psi`, mientras que `IHphi` y `IHpsi` serán los nombres de sus respectivas hipótesis inductivas.
- `phi` será la fórmula de menor tamaño del caso inductivo `Diamond phi`, mientras que `IH` será el nombre de su hipótesis inductiva.
- `d` será un elemento de `Dyn`, `phi` será la fórmula de menor tamaño del caso inductivo `DynDiamond d phi`, mientras que `IH` será el nombre de su hipótesis inductiva.

Ahora continuemos con cada caso.

▷ `phi = Atom p`

Debemos demostrar que:

```
forall (w : W) (w' : W')
  (R : W → W → Prop) (R' : W' → W' → Prop)
  (Z : model_to_model_relation W W'),
atomic_harmony W L W' L' Z →
zig W W' Z →
zag W W' Z →
(forall d : Dyn, f_zig W W' Z (Semantics.F d)) →
(forall d : Dyn, f_zag W W' Z (Semantics.F d)) →
Z w R w' R' →
# W, R, L >> w |= Atom p ↔ # W', R', L' >> w' |= Atom p
```

La demostración es sencilla. Primero aplicamos la definición de `satisfies` y luego introducimos todas las variables cuantificadas y las hipótesis:

```
unfold satisfies.
intros w w' S S' Z HAtomicHarmony HZig HZag HFZig HFZag HZwSw'S'.
```

En este punto nos queda demostrar que:

```
L w p ↔ L' w' p
```

Pero sabemos que por armonía atómica esto debe ser cierto:

```
rewrite (HAtomicHarmony w S w' S' HZwSw'S').
tauto.
```

```
▷ phi = Bottom
```

Debemos demostrar que:

```
forall (w : W) (w' : W')
  (R : W → W → Prop) (R' : W' → W' → Prop)
  (Z : model_to_model_relation W W'),
atomic_harmony W L W' L' Z →
zig W W' Z →
zag W W' Z →
(forall d : Dyn, f_zig W W' Z (Semantics.F d)) →
(forall d : Dyn, f_zag W W' Z (Semantics.F d)) →
Z w R w' R' →
# W, R, L >> w |= Bottom ↔ # W', R', L' >> w' |= Bottom
```

En este caso basta hacer:

```
tauto.
```

```
▷ phi = If phi psi
```

Debemos demostrar que:

```
forall (w : W) (w' : W')
  (R : W → W → Prop) (R' : W' → W' → Prop)
  (Z : model_to_model_relation W W'),
atomic_harmony W L W' L' Z →
zig W W' Z →
zag W W' Z →
(forall d : Dyn, f_zig W W' Z (Semantics.F d)) →
(forall d : Dyn, f_zag W W' Z (Semantics.F d)) →
```

```
Z w R w' R' →
# W, R, L >> w |= (phi →' psi) ↔
# W', R', L' >> w' |= (phi →' psi)
```

Primero aplicamos la definición de `satisfies` pero teniendo el cuidado de no aplicarla muchas veces. Para esto hacemos:

```
unfold satisfies. fold satisfies.
```

Luego introducimos las siguientes hipótesis:

```
w : W
w' : W'
S : W → W → Prop
S' : W' → W' → Prop
Z : model_to_model_relation W W'
HAtomicHarmony : atomic_harmony W L W' L' Z
HZig : zig W W' Z
HZag : zag W W' Z
HFZig : forall d : Dyn, f_zig W W' Z (Semantics.F d)
HFZag : forall d : Dyn, f_zag W W' Z (Semantics.F d)
HZwSw'S' : Z w S w' S'
```

con la táctica:

```
intros w w' S S' Z HAtomicHarmony HZig HZag HFZig HFZag HZwSw'S'.
```

El subobjetivo queda:

```
(# W, S, L >> w |= phi → # W, S, L >> w |= psi) ↔
(# W', S', L' >> w' |= phi → # W', S', L' >> w' |= psi)
```

Por lo que usaremos la táctica `split` para realizar un análisis por casos de cada implicación.

Como las demostraciones de las dos implicaciones son idénticas veremos únicamente la implicación de izquierda a derecha y el lector podrá confirmar fácilmente que la otra es idéntica. La demostración completa se puede encontrar en el Capítulo F del apéndice.

Mediante la siguiente táctica:

```
intros H Hsat.
```

introducimos las siguientes hipótesis:

```
H : # W, S, L >> w |= phi → # W, S, L >> w |= psi
Hsat : # W', S', L' >> w' |= phi
```

por lo que nos queda demostrar:

```
# W', S', L' >> w' |= psi
```

Usaremos la hipótesis inductiva de `psi` que nos dice que para demostrar `# W', S', L' >> w' |= psi` basta probar `# W, S, L >> w |= psi`. Luego aplicaremos la hipótesis `H` que reducirá el objetivo a demostrar `# W, S, L >> w |= phi`. Pero la hipótesis inductiva de `phi` afirma que para demostrar esto es suficiente demostrar `# W', R', L' >> w' |= phi`, algo que sabemos que es cierto por `Hsat`. Teniendo en cuenta lo anterior, la prueba quedaría:

```
apply (IHpsi w w' S S' Z
           HAtomicHarmony HZig HZag HFZig HFZag HZwSw'S').
apply H.
apply (IHphi w w' S S' Z
           HAtomicHarmony HZig HZag HFZig HFZag HZwSw'S').
apply Hsat.
```

Como la implicación de izquierda a derecha y la implicación de derecha a izquierda son idénticas podemos escribir la demostración de la siguiente manera:

```
unfold satisfies; fold satisfies.
intros w w' S S' Z HAtomicHarmony HZig HZag HFZig HFZag HZwSw'S'.
split;
  intros H Hsat;
  apply (IHpsi w w' S S' Z
             HAtomicHarmony HZig HZag HFZig HFZag HZwSw'S');
  apply H;
  apply (IHphi w w' S S' Z
             HAtomicHarmony HZig HZag HFZig HFZag HZwSw'S');
  apply Hsat.
```

De esta manera se aplican todas las tácticas a ambos subobjetivos (es decir, a ambas implicaciones).

```
▷ phi = Diamond psi
```

En este caso debemos demostrar que:

```
forall (w : W) (w' : W')
  (R : W → W → Prop) (R' : W' → W' → Prop)
  (Z : model_to_model_relation W W'),
atomic_harmony W L W' L' Z →
zig W W' Z →
zag W W' Z →
(forall d : Dyn, f_zig W W' Z (Semantics.F d)) →
(forall d : Dyn, f_zag W W' Z (Semantics.F d)) →
Z w R w' R' →
# W, R, L >> w |= (<m> phi) ↔ # W', R', L' >> w' |= (<m> phi)
```

Análogamente al caso anterior empezamos con las siguientes tácticas:

```
unfold satisfies; fold satisfies.
intros w w' S S' Z HAtomicHarmony HZig HZag HFZig HFZag HZwSw'S'.
```

El objetivo a demostrar nos queda:

```
(exists v : W, S w v ∧ # W, S, L >> v |= phi) ↔
(exists v : W', S' w' v ∧ # W', S', L' >> v |= phi)
```

Como son dos casos, los analizamos por separado con la táctica `split`. Debido a que ambos casos son completamente análogos aquí sólo presentaremos la demostración de uno de ellos, la demostración completa puede verse en el Capítulo F del apéndice.

Demostraremos el implica de izquierda a derecha. Mediante la táctica:

```
intro H.
```

introducimos el antecedente del implica con el nombre `H`:

```
exists v : W', S' w' v ∧ # W', S', L' >> v |= phi
```

A continuación tomaremos un ejemplo de este existencial que denominaremos `v`. Por `H`, sabemos que `v` tiene tipo `W` y que satisface dos propiedades que nombraremos `HSwv` y `Hsatv`:

```
HSwv : S w v
Hsatv : # W, S, L >> v |= phi
```

Para lograr esto basta hacer:

```
destruct H as [v [HSwv Hsatv]].
```

Como nos interesa probar que:

```
exists v' : W', S' w' v' ∧ # W', S', L' >> v' |= phi
```

vamos a aplicar la propiedad zig:

```
HZig : forall (w : W) (S : W → W → Prop) (w' : W')
         (S' : W' → W' → Prop) (v : W),
       Z w S w' S' → S w v → exists v' : W', Z v S v' S' ∧ S' w' v'
```

con los argumentos apropiados en `HSwv` para obtener:

```
exists v' : W', Z v S v' S' ∧ S' w' v'
```

Notemos que hacer esto nos permitirá dar un ejemplo de un `v'` que cumple `S' w' v'`. Una vez que hagamos esto, tendremos que demostrar que ese mismo `v'` cumple `# W', S', L' >> v' |= phi`. Para esto último utilizaremos la hipótesis inductiva y el hecho de que `Z v S v' S'`.

Como dijimos, primero tomamos un ejemplo del cuantificador existencial que llamaremos `v'` y nombraremos a las dos propiedades que por `Hzig` sabemos que posee `v'` de la siguiente manera:

```
HZvv' : Z v S v' S'
HS'w'v' : S' w' v'
```

Todo esto lo podemos lograr haciendo:

```
apply (HZig w S w' S' v HZwSw'S') in HSwv as [v' [HZvv' HS'w'v']].
```

Ahora tomamos el `v'` como nuestro candidato para demostrar el existencial:

```
exists v'.
```

por lo que sólo nos queda probar:

```
R' w' v' ∧ # W', R', L' >> v' |= phi
```

Para esto, separamos en dos casos usando la siguiente táctica:

```
split.
```

Notemos que ya sabemos que `S' w' v'` por la hipótesis `HS'w'v'`, por lo que basta con usar la táctica `assumption`. para este caso. Para el otro caso, basta aplicar la hipótesis inductiva `IH` con los argumentos adecuados para reducir el subobjetivo a `# W', R', L' >> v' |= phi`, algo que podemos demostrar usando `assumption`., puesto que `Hsatv' : # W', R', L' >> v' |= phi` es una de nuestras hipótesis.

Por lo tanto, este análisis por casos quedaría:

```
* assumption.
* apply (IH v v' S S' Z
           HAtomicHarmony HZig HZag HFZig HFZag HZvv').
  assumption.

▷ phi = DynDiamond d phi
```

Para este caso debemos demostrar:

```
forall (w : W) (w' : W') (R : W → W → Prop)
  (R' : W' → W' → Prop) (Z : model_to_model_relation W W'),
atomic_harmony W L W' L' Z →
zig W W' Z →
zag W W' Z →
(forall d0 : Dyn, f_zig W W' Z (Semantics.F d0)) →
(forall d0 : Dyn, f_zag W W' Z (Semantics.F d0)) →
Z w R w' R' →
# W, R, L >> w |= (<o> d phi) ↔ # W', R', L' >> w' |= (<o> d phi)
```

Análogamente al caso anterior empezamos con las siguientes tácticas:

```
unfold satisfies; fold satisfies.
intros w w' S S' Z HAtomicHarmony HZig HZag HFZig HFZag HZwSw'S'.
```

El objetivo a demostrar nos queda:

```
(exists (v : W) (R' : W → W → Prop),
   Semantics.F d W w S v R' ∧ # W, R', L >> v |= phi) ↔
(exists (v : W') (R' : W' → W' → Prop),
   Semantics.F d W' w' S' v R' ∧ # W', R', L' >> v |= phi)
```

Como son dos casos, los analizamos por separado con la táctica `split`. Debido a que ambos casos son completamente análogos aquí sólo presentaremos la demostración de uno de ellos, la demostración completa puede verse en el Capítulo F del apéndice.

Demostraremos el implica de izquierda a derecha. Mediante la táctica:

```
intro H.
```

introducimos el antecedente del implica con el nombre `H`:

```
exists (v : W) (T : W → W → Prop),
  Semantics.F d W w S v T ∧ # W, T, L >> v |= phi
```

A continuación tomaremos un ejemplo de este existencial que denominaremos `v` y un ejemplo del segundo existencial que denominaremos `T`. Por `H`, sabemos que `v` tiene tipo `W`, que `T` tiene tipo `W → W → Prop` y que satisfacen las siguientes dos propiedades que nombraremos `HfWwSvT` y `HsatTv`:

```
HfWwSvT : Semantics.F d W w S v T
HsatTv : # W, T, L >> v |= phi
```

Para lograr esto basta hacer:

```
destruct H as [v [T [HfWwSvT HsatTv]]].
```

Como nos interesa probar que:

```
exists (v' : W') (T' : W' → W' → Prop),
  Semantics.F d W' w' S' v' T' ∧ # W', T', L' >> v' |= phi
```

vamos a aplicar la propiedad `HFzig` con los argumentos apropiados en `HfWwSvT` para obtener:

```
HfWwSvT : exists (v' : W') (T' : W' → W' → Prop),
          Semantics.F d W' w' S' v' T' ∧ Z v T v' T'
```

Notemos que hacer esto nos permitirá dar un ejemplo de un `v'` y de un `T'` que cumplan `Semantics.F d W' w' S' v' T'`. Una vez que hagamos esto, tendremos que demostrar que esos mismos `v'` y `T'` cumplen `# W', T', L' >> v' |= phi`

Para esto último utilizaremos la hipótesis inductiva y el hecho de que `Z v T v' T'`.

Como dijimos, primero tomamos un ejemplo del cuantificador existencial que llamaremos `v'` y nombraremos a las dos propiedades que por `Hzig` sabemos que posee `v'` de la siguiente manera:

```
HfW'w'S'v'T' : Semantics.F d W' w' S' v' T'
HZvTv'T' : Z v T v' T'
```

Todo esto lo podemos lograr haciendo:

```
apply (HFZig d w S w' S' v T HZwSw'S') in HfWwSvT
  as [v' [T' [HfW'w'S'v'T' HZvTv'T']]].
```

Ahora tomamos el `v'` y `T'` como nuestros candidatos para demostrar el existencial:

```
exists v'. exists T'.
```

por lo que sólo nos queda probar:

```
Semantics.F d W' w' S' v' T' ∧ # W', T', L' >> v' |= phi
```

Para esto, separamos en dos casos usando la siguiente táctica:

```
split.
```

Notemos que ya sabemos que `Semantics.F d W' w' S' v' T'` por la hipótesis `HfW'w'S'v'T'`, por lo que basta con usar la táctica `assumption`. para este caso. Para el otro caso, basta aplicar la hipótesis inductiva `IH` con los argumentos adecuados para reducir el subobjetivo a `# W, T, L >> v |= phi`, algo que podemos demostrar usando `assumption.`, puesto que `HsatTv : # W, T, L >> v |= phi` es una de nuestras hipótesis.

Por lo tanto, este análisis por casos quedaría:

```
* assumption.
* apply (IH v v' T T' Z HAtomicHarmony HZig
          HZag HFZig HFZag HZvTv'T').
  assumption.
```

Parte III

CONCLUSIÓN

7 CONCLUSIÓN

En la primera parte de nuestro trabajo, estudiamos la sintaxis y la semántica, primero de la lógica modal básica, y luego de ciertas lógicas dinámicas modales desarrolladas en [3]. Una vez hecho esto, describimos brevemente las características de Coq relevantes para poder formalizar estas lógicas, demostrar los resultados estudiados y finalmente verificarlos. En la segunda parte del trabajo, formalizamos utilizando Coq las lógicas estudiadas basándonos en el trabajo de [33]. Para ambas lógicas formalizamos la sintaxis de los lenguajes y todas las nociones semánticas necesarias para poder enunciar el teorema de invarianza bajo bisimulación. Luego formalizamos, demostramos y verificamos este teorema para cada una de las lógicas. La principal contribución de nuestro trabajo ha sido la formalización de las lógicas modales dinámicas y la demostración y verificación de los teoremas de invarianza bajo bisimulación para la lógica modal básica y para las lógicas modales dinámicas.

En el Capítulo 2 estudiamos la sintaxis y la semántica de la lógica modal básica. En particular, dimos la sintaxis del lenguaje, definimos los modelos de Kripke, la noción de satisfacibilidad de una fórmula sobre un modelo, el concepto de bisimulación y el de equivalencia modal y finalmente enunciamos y demostramos informalmente el teorema de invarianza bajo bisimulación. En el Capítulo 3 introducimos los conceptos análogos para las lógicas modales dinámicas, esto implicó la introducción de nuevos conceptos, propios de estas lógicas modales. Finalmente también dimos una demostración detallada del teorema de invarianza bajo bisimulación para estas lógicas dinámicas.

En el Capítulo 4 estudiamos algunas de las características más relevantes de Coq. Dado que Coq puede verse tanto como un lenguaje de programación funcional como una lógica intuicionista, en este capítulo tuvimos que estudiar ambos enfoques. Primero presentamos a Coq como un lenguaje funcional con tipos dependientes mostrando cómo definir tipos de datos, funciones, tipos de datos paramétricos, funciones recursivas, tipos polimórficos y tipos dependientes. Luego presentamos a Coq como un lenguaje para formalizar y demostrar enunciados matemáticos, esto es, como una lógica. También mostramos cómo utilizar el lenguaje de tácticas, que permite automatizar la construcción de las demostraciones matemáticas en Coq.

En el Capítulo 5 dimos la formalización de la lógica modal básica en Coq. Esto implicó por un lado definir formalmente la sintaxis de la lógica modal básica y por el otro, formalizar todas las nociones semánticas como bisimulación, equivalencia modal, modelo de Kripke, etc. Luego enunciamos, demostramos y verificamos el teorema de invarianza bajo bisimulación para la lógica modal básica utilizando el lenguaje de tácticas Ltac. En el Capítulo 6 dimos una formalización análoga para las lógicas modales dinámicas en Coq. Igual que en el capítulo anterior, logramos verificar el teorema de invarianza bajo bisimulación para las lógicas modales dinámicas.

7.1 TRABAJOS RELACIONADOS Y FUTUROS

Hasta donde sabemos, la tesis de de Wind [33] es la primera formalización de la lógica modal básica en Coq. En esta tesis, además de la formalización de la lógica modal básica, se presentaron formalizaciones para las lógicas modales S5 y $S5^n$. Para la lógica modal básica, se formalizaron los modelos de Kripke y el sistema de deducción natural. Para las lógicas S5 y $S5^n$, un sistema de deducción natural también fue formalizado. Luego la autora utiliza estas formalizaciones para resolver formalmente dos rompecabezas, el *wise men puzzle* y el *muddy children puzzle*. En nuestro trabajo utilizamos la formalización de la lógica modal básica dada en la tesis de de Wind y la extendimos a las lógicas modales dinámicas. Además de esto, formalizamos el concepto de bisimulación, que no estaba presente en [33] y verificamos el teorema de invarianza bajo bisimulación tanto para la lógica modal básica como para la lógica modal dinámica.

Existen otros trabajos en donde también se demuestran meta teoremas para lógicas no clásicas. En [35] se presenta una formalización de la lógica lineal en Coq, una mecanización de la demostración del teorema de *cut-elimination* y la completitud de *focusing* para la lógica lineal y una codificación de la lógica intuicionista proposicional en la lógica lineal formalizada. Otro ejemplo es [13], en donde los autores dan una formalización del teorema de correspondencia de Sahlqvist [31] para fórmulas muy simples (*very simple Sahlqvist formulae*) en Coq. A partir de esta formalización los autores pudieron extraer un programa verificado de Haskell que computa los correspondientes de primer orden para cualquier fórmula modal muy simple de Sahlqvist. En ningún trabajo anterior se había presentado un algoritmo verificado para hacer esta tarea.

En todos estos trabajos las formalización de la sintaxis y la semántica de los lenguajes modales siempre se realiza nuevamente puesto que no existen hasta donde sabemos, librerías en Coq para lógicas modales. Trabajos futuros se podrían beneficiar de la existencia de librerías que formalicen los resultados más importantes de lógicas modales. Este trabajo es el primer paso en la construcción de una librería para lógicas no clásicas en Coq. Hay mucho por delante.

Como vimos, existen varias formalizaciones de resultados sobre lógicas modales. Trabajos futuros se podrían beneficiar de la existencia de librerías que formalicen los resultados más importantes de lógicas modales. Un primer paso para lograr este objetivo es unificar todas las formalizaciones existentes de lógicas modales en Coq en una única librería con una interfaz consistente.

Es muy frecuente definir nuevas lógicas modales agregando nuevos operadores modales y dándoles semántica, de acuerdo a las necesidades particulares. Un problema que surge es que muchas veces es necesario volver a demostrar resultados como completitud, invarianza bajo bisimulación, correctitud, etc. debido a la incorporación de estos operadores. Pero estas demostraciones tienen estructuras similares. Nuevas tácticas de Coq podrían ser utilizadas para explotar esta similitud estructural entre las demostraciones y asistir a los investigadores en la tarea de crear nuevas lógicas modales.

En cuanto a la formalización de lógicas dinámicas, existen otros resultados que pueden ser formalizados. Por ejemplo, en [3] se presentan traducciones estándar a primer orden de lógicas dinámicas con ciertos operadores dinámicos específicos

y una traducción estándar a segundo orden para cualquier lógica dinámica con operadores que modifican la relación de accesibilidad. La formalización de estos resultados podría permitir, por un lado verificar que estas traducciones son correctas y por el otro extraer programas certificados que realicen estas traducciones.

Parte IV

APÉNDICE

A SINTÁXIS DE LÓGICA MODAL BÁSICA EN COQ

```
(* A set of propositional symbols *)

Inductive prop : Set :=
p : nat → prop.

(* The set of formulas defined over prop *)

Inductive form : Set :=
| Atom   : prop → form
| Bottom : form
| If     : form → form → form
| Diam   : form → form.

(* Syntactic sugar of other common operators *)

Definition Not (phi : form) : form :=
If phi Bottom.

Definition Top : form :=
Not Bottom.

Definition And (phi psi : form) : form :=
Not (If phi (Not psi)).

Definition Or (phi psi : form) : form :=
If (Not phi) psi.

Definition Iif (phi psi : form) : form :=
And (If phi psi) (If psi phi).

Definition Box (phi : form) : form :=
Not (Diam (Not phi)).

(* Notation *)

Notation "p ∧' q" := (And p q)
                     (at level 80, right associativity).

Notation "p ∨' q" := (Or p q)
                     (at level 85, right associativity).

Notation "~' p" := (Not p)
                   (at level 70, right associativity).

Notation "p →' q" := (If p q)
                     (at level 90, right associativity).
```

```
Notation "p ↔ ' q" := (Iif p q)
                        (at level 95, right associativity).

Notation "[] F" := (Box F)
                       (at level 65, right associativity).

Notation "<> F" := (Diam F)
                      (at level 65, right associativity).
```

B | SEMÁNTICA DE LÓGICA MODAL BÁSICA EN COQ

```
From ModalLogic Require Import Syntax.

Fixpoint satisfies
        (W : Set)
        (R : W → W → Prop)
        (V : W → prop → Prop)
        (w : W)
        (phi : form) : Prop :=
match phi with
| Atom v ⇒ (V w v)
| Bottom ⇒ False
| If phi psi ⇒ satisfies W R V w phi → satisfies W R V w psi
| Diam phi ⇒ exists v : W, R w v ∧ satisfies W R V v phi
end.

Notation "# W , R , L >> w |= phi" := (satisfies W R L w phi)
                                     (at level 30).

Definition equivalent_at_points W R L W' R' L' w w' :=
  forall (phi : form), (# W , R , L >> w |= phi) ↔
                 (# W' , R' , L' >> w' |= phi).

Definition atomic_harmony
          (W : Set) (V : W → prop → Prop)
          (W' : Set) (V' : W' → prop → Prop)
          (Z : W → W' → Prop) : Prop :=
  forall w w',
  Z w w' → V w = V' w'.

Definition zig
          (W : Set) (R : W → W → Prop)
          (W' : Set) (R' : W' → W' → Prop)
          (Z : W → W' → Prop) : Prop :=
  forall w w' v, Z w w' → R w v →
    (exists v' : W', Z v v' ∧ R' w' v').

Definition zag
          (W : Set) (R : W → W → Prop)
          (W' : Set) (R' : W' → W' → Prop)
          (Z : W → W' → Prop) : Prop :=
  forall w w' v', Z w w' → R' w' v' → (exists v : W, Z v v' ∧ R w v).

Definition bisimulation W R L W' R' L' Z :=
  (atomic_harmony W L W' L' Z) ∧ (zig W R W' R' Z) ∧ (zag W R W' R' Z).

Definition bisimulation_at_points W R L W' R' L' Z w w' :=
```

```
  bisimulation W R L W' R' L' Z ∧ Z w w'.

Definition bisimulable W R L W' R' L' :=
  exists Z, bisimulation W R L W' R' L' Z.

Definition bisimulable_at_points W R L W' R' L' w w':=
  exists Z, bisimulation W R L W' R' L' Z ∧ Z w w'.
```

C | TEOREMA DE INVARIANCIA DE LA LÓGICA MODAL BÁSICA EN COQ

```
From ModalLogic Require Import Syntax Semantics.

Theorem InvarianceUnderBisimulation :
  forall W R L W' R' L' w w',
    bisimulable_at_points W R L W' R' L' w w' →
    equivalent_at_points W R L W' R' L' w w'.
Proof.
  intros W R L W' R' L' w w'.

  unfold bisimulable_at_points.
  unfold equivalent_at_points.
  unfold bisimulation.

  intro H.

  destruct H as [Z H].
  destruct H as [H HZww'].
  destruct H as [HAtomicHarmony H].
  destruct H as [HZig HZag].

  intro phi.

  generalize dependent w'.
  generalize dependent w.

  induction phi as [p | | phi IHphi psi IHpsi | phi IH].

  - (* Atom *)
    unfold satisfies. intros w w' HZww'.
    rewrite (HAtomicHarmony w w' HZww'); tauto.

  - (* Bottom *)
    tauto.

  - (* If *)
    unfold satisfies. fold satisfies. intros w w' HZww'.
    split;
      intros H Hsat;
      apply (IHpsi w w' HZww');
      apply H;
      apply (IHphi w w' HZww');
      apply Hsat.

  - (* Modal *)
    unfold satisfies. fold satisfies. intros w w' HZww'.
    split.
```

```
    + intro H.
      destruct H as [v [HRwv Hsatv]].
      unfold zig in HZig.
      apply (HZig w w' v HZww') in HRwv as [v' [HZvv' HR'w'v']].
      exists v'.
      split.
      * assumption.
      * apply (IH v v' HZvv'). assumption.
    + intro H.
      destruct H as [v' [HR'w'v' Hsatv']].
      unfold zag in HZag.
      apply (HZag w w' v' HZww') in HR'w'v' as [v [HZvv' HRwv]].
      exists v.
      split.
      * assumption.
      * apply (IH v v' HZvv'). assumption.
Qed.
```

D | SINTÁXIS DE LÓGICA MODAL DINÁMICA EN COQ

```
Variable Dyn : Set.

(* A set of propositional symbols *)
Inductive prop: Set :=
p : nat → prop.

(* The set of formulas defined over prop *)
Inductive form : Type :=
| Atom   : prop → form
| Bottom : form
| If     : form → form → form
| Diam   : form → form
| DynDiam : Dyn → form → form.

(* Syntactic sugar *)
Definition Not (phi : form) : form :=
If phi Bottom.

Definition Top : form :=
Not Bottom.

Definition And (phi psi : form) : form :=
Not (If phi (Not psi)).

Definition Or (phi psi : form) : form :=
If (Not phi) psi.

Definition Iif (phi psi : form) : form :=
And (If phi psi) (If psi phi).

Definition Box (phi : form) : form :=
Not (Diam (Not phi)).

Definition DynBox (d : Dyn) (phi : form) : form :=
Not (DynDiam d (Not phi)).


(* Notation *)

Notation "p ∧' q" := (And p q)
                     (at level 80, right associativity).

Notation "p ∨' q" := (Or p q)
                     (at level 85, right associativity).

Notation "~' p" := (Not p)
```

```
                    (at level 70, right associativity).

Notation "p → ' q" := (If p q)
                      (at level 90, right associativity).

Notation "p ↔ ' q" := (Iif p q)
                      (at level 95, right associativity).

Notation "[m] phi" := (Box phi)
                       (at level 65, right associativity).

Notation "<m> phi" := (Diam phi)
                       (at level 65, right associativity).

Notation "<o> d phi" := (DynDiam d phi)
                         (at level 65, right associativity).

Notation "[o] d phi" := (DynBox d phi)
                         (at level 65, right associativity).
```

E SEMÁNTICA DE LÓGICA MODAL DINÁMICA EN COQ

```
From ModalLogic Require Import Syntax.

Definition model_update_function : Type := forall (W : Set),
  let R := (W → W → Prop) in W → R → (W → R → Prop).

Definition muf := model_update_function.

Definition model_to_model_relation W W' : Type :=
  W → (W → W → Prop) → W' → (W' → W' → Prop) → Prop.

Definition Dyn := Syntax.Dyn.

Variable F : Dyn → muf.

Fixpoint satisfies
         (W :  Set)
         (R :  W → W → Prop)
         (L :  W → prop → Prop)
         (w :  W)
         (phi :  form) :  Prop :=
match phi with
| Atom v ⇒ (L w v)
| Bottom ⇒ False
| phi1 → ' phi2 ⇒ (satisfies W R L w phi1) → (satisfies W R L w phi2)
| Diam phi ⇒
    exists v : W,  R w v ∧ (satisfies W R L v phi)
| DynDiam d phi ⇒
  let fw := F d W in
  exists (v :  W) (R' :  W → W → Prop), fw w R v R' ∧
                            satisfies W R' L v phi
end.

Notation "# W , R , L >> w |= phi" := (satisfies W R L w phi)
                                      (at level 30).

Definition equivalent_at_points W R L W' R' L' w w' :=
  forall (phi:form), (# W ,  R ,  L >> w |= phi) ↔
               (# W' ,  R' ,  L' >> w' |= phi).

Definition atomic_harmony
           (W :  Set) (L :  W → prop → Prop)
           (W' :  Set) (L' :  W' → prop → Prop)
           (Z :  model_to_model_relation W W') : Prop :=
  forall w S w' S',
  Z w S w' S' → L w = L' w'.
```

```
Definition zig
           (W : Set)
           (W' : Set)
           (Z : model_to_model_relation W W') : Prop :=
  forall w S w' S' v, Z w S w' S' →
    S w v → (exists v' : W', Z v S v' S' ∧ S' w' v').

Definition zag
           (W : Set)
           (W' : Set)
           (Z : model_to_model_relation W W') : Prop :=
  forall w S w' S' v', Z w S w' S' →
    S' w' v' → (exists v : W, Z v S v' S' ∧ S w v).

Definition f_zig
           (W : Set)
           (W' : Set)
           (Z : model_to_model_relation W W')
           (f : muf) : Prop :=
  let (fw, fw') := (f W, f W') in
  forall w S w' S' v T, Z w S w' S' →
    fw w S v T → (exists (v' : W') T', fw' w' S' v' T' ∧ Z v T v' T').

Definition f_zag
           (W : Set)
           (W' : Set)
           (Z : model_to_model_relation W W')
           (f : muf) : Prop :=
  let (fw, fw') := (f W, f W') in
  forall w S w' S' v' T', Z w S w' S' →
    fw' w' S' v' T' → (exists (v : W) T, fw w S v T ∧ Z v T v' T').

Definition bisimulation W L W' L' Z :=
  (atomic_harmony W L W' L' Z) ∧ (zig W W' Z) ∧ (zag W W' Z) ∧
  (forall d : Dyn, (f_zig W W' Z (F d))) ∧
  forall d : Dyn, (f_zag W W' Z (F d)).

Definition bisimulation_at_points W R L W' R' L' Z w w' :=
  bisimulation W L W' L' Z ∧ Z w R w' R'.

Definition bisimulable W L W' L' :=
  exists Z, bisimulation W L W' L' Z.

Definition bisimulable_at_points W R L W' R' L' w w':=
  exists Z, bisimulation W L W' L' Z ∧ Z w R w' R'.
```

F | TEOREMA DE INVARIANCIA DE LA LÓGICA MODAL DINÁMICA EN COQ

```
From ModalLogic Require Import Syntax Semantics.

Theorem InvarianceUnderBisimulation :
  forall W R L W' R' L' w w',
    bisimulable_at_points W R L W' R' L' w w' →
    equivalent_at_points W R L W' R' L' w w'.
Proof.
  intros W R L W' R' L' w w'.

  unfold bisimulable_at_points.
  unfold equivalent_at_points.
  unfold bisimulation.

  intro H.

  destruct H as [Z H].
  destruct H as [H HZwSw'S'].
  destruct H as [HAtomicHarmony H].
  destruct H as [HZig H].
  destruct H as [HZag H].
  destruct H as [HFZig HFZag].

  intro phi.

  generalize dependent Z.
  generalize dependent R'.
  generalize dependent R.
  generalize dependent w'.
  generalize dependent w.

  induction phi as [p | | phi IHphi psi IHpsi | phi IH | d phi IH].

  + (* Atom *)
    unfold satisfies.
    intros w w' S S' Z HAtomicHarmony HZig HZag HFZig HFZag HZwSw'S'.
    rewrite (HAtomicHarmony w S w' S' HZwSw'S').
    tauto.

  + (* Bottom *)
    tauto.

  + (* If *)
    unfold satisfies; fold satisfies.
    intros w w' S S' Z HAtomicHarmony HZig HZag HFZig HFZag HZwSw'S'.
    split;
      intros H Hsat;
```

```
    apply (IHpsi w w' S S' Z
                HAtomicHarmony HZig HZag HFZig HFZag HZwSw'S');
    apply H;
    apply (IHphi w w' S S' Z
                HAtomicHarmony HZig HZag HFZig HFZag HZwSw'S');
    apply Hsat.

+ (* Modal *)
  unfold satisfies; fold satisfies.
  intros w w' S S' Z HAtomicHarmony HZig HZag HFZig HFZag HZwSw'S'.
  split.
  - intro H.
    destruct H as [v [HSwv Hsatv]].
    unfold zig in HZig.
    apply (HZig w S w' S' v HZwSw'S') in HSwv as [v' [HZvv' HS'w'v']].
    exists v'.
    split.
    * assumption.
    * apply (IH v v' S S' Z
                HAtomicHarmony HZig HZag HFZig HFZag HZvv').
      assumption.
  - intro H.
    unfold zag in HZag.
    destruct H as [v' H].
    destruct H as [HSw'v' Hsatv'].
    apply (HZag w S w' S' v' HZwSw'S') in HSw'v' as Hexists.
    destruct Hexists as [v Hexists].
    destruct Hexists as [HZvv' HSwv].
    exists v.
    split.
    * assumption.
    * apply (IH v v' S S' Z
                HAtomicHarmony HZig HZag HFZig HFZag HZvv').
      assumption.

+ (* Dynamic *)
  unfold satisfies; fold satisfies.
  intros w w' S S' Z HAtomicHarmony HZig HZag HFZig HFZag HZwSw'S'.
  split.
  - unfold satisfies; fold satisfies.
    intro H.
    destruct H as [v [T [HfWwSvT HsatTv]]].

    apply (HFZig d w S w' S' v T HZwSw'S') in HfWwSvT
      as [v' [T' [HfW'w'S'v'T' HZvTv'T']]].

    exists v'. exists T'.

    split.
      * assumption.
      * apply (IH v v' T T' Z HAtomicHarmony HZig
                  HZag HFZig HFZag HZvTv'T').
        assumption.
```

```
  - unfold satisfies; fold satisfies.
    intro H.
    destruct H as [v' [T' [HfW'w'S'v'T' HsatT'v']]].

    apply (HFZag d w S w' S' v' T' HZwSw'S') in HfW'w'S'v'T'
      as [v [T [HfWwSvT HZvTv'T']]].

    exists v. exists T.

    split.
      * assumption.
      * apply (IH v v' T T' Z HAtomicHarmony HZig
                  HZag HFZig HFZag HZvTv'T').
        assumption.

Qed.
```

BIBLIOGRAFÍA

[1] Carlos Areces, Raul Fervari y Guillaume Hoffmann. «Moving Arrows and Four Model Checking Results». En: *Logic, Language, Information and Computation - 19th International Workshop, WoLLIC 2012, Buenos Aires, Argentina, September 3-6, 2012. Proceedings*. 2012, págs. 142-153. DOI: 10.1007/978-3-642-32621-9_11. URL: https://doi.org/10.1007/978-3-642-32621-9_11 (vid. págs. 5, 17).

[2] Carlos Areces, Raul Fervari y Guillaume Hoffmann. «Swap logic». En: *Logic Journal of the IGPL* 22.2 (2014), págs. 309-332. DOI: 10.1093/jigpal/jzt030. URL: https://doi.org/10.1093/jigpal/jzt030 (vid. pág. 17).

[3] Carlos Areces, Raul Fervari y Guillaume Hoffmann. «Relation-changing modal operators». En: *Logic Journal of the IGPL* 23.4 (2015), págs. 601-627. DOI: 10.1093/jigpal/jzv020. URL: https://doi.org/10.1093/jigpal/jzv020 (vid. págs. 3, 17, 21, 22, 65, 66).

[4] Carlos Areces, Raul Fervari, Guillaume Hoffmann y Mauricio Martel. «Relation-Changing Logics as Fragments of Hybrid Logics». En: *Proceedings of the Seventh International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2016, Catania, Italy, 14-16 September 2016*. 2016, págs. 16-29. DOI: 10.4204/EPTCS.226.2. URL: https://doi.org/10.4204/EPTCS.226.2 (vid. pág. 17).

[5] Carlos Areces, Raul Fervari, Guillaume Hoffmann y Mauricio Martel. «Undecidability of Relation-Changing Modal Logics». En: *Dynamic Logic. New Trends and Applications - First International Workshop, DALI 2017, Brasilia, Brazil, September 23-24, 2017, Proceedings*. 2017, págs. 1-16. DOI: 10.1007/978-3-319-73579-5_1. URL: https://doi.org/10.1007/978-3-319-73579-5_1 (vid. pág. 17).

[6] Carlos Areces, Raul Fervari, Guillaume Hoffmann y Mauricio Martel. «Satisfiability for relation-changing logics». En: *J. Log. Comput.* 28.7 (2018), págs. 1443-1470. DOI: 10.1093/logcom/exy022. URL: https://doi.org/10.1093/logcom/exy022 (vid. pág. 17).

[7] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau y Bas Spitters. «The HoTT library: a formalization of homotopy type theory in Coq». En: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*. 2017, págs. 164-172. DOI: 10.1145/3018610.3018615. URL: https://doi.org/10.1145/3018610.3018615 (vid. pág. 4).

[8] Sophie Bernard, Yves Bertot, Laurence Rideau y Pierre-Yves Strub. «Formal proofs of transcendence for e and pi as an application of multivariate and symmetric polynomials». En: *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January*

20-22, 2016. 2016, págs. 76-87. DOI: 10.1145/2854065.2854072. URL: https://doi.org/10.1145/2854065.2854072 (vid. pág. 4).

[9] Yves Bertot y Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. 1st. Springer Publishing Company, Incorporated, 2010 (vid. págs. 4, 5).

[10] P. Blackburn y J. van Benthem. «Modal Logic: A Semantic Perspective». En: *Handbook of Modal Logic*. Elsevier North-Holland, 2006 (vid. págs. 3, 8).

[11] P. Blackburn, M. de Rijke e Y. Venema. *Modal Logic*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2002. ISBN: 9781316101957 (vid. págs. 14, 15).

[12] Ana Bove, Peter Dybjer y Ulf Norell. «A brief overview of Agda–a functional language with dependent types». En: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2009, págs. 73-78 (vid. pág. 4).

[13] Caitlin D'Abrera y Rajeev Goré. «Verified Synthesis of (very simple) Sahlqvist Correspondents Via Coq». En: *AiML 2018* (), pág. 26 (vid. pág. 66).

[14] David Delahaye. «A Tactic Language for the System Coq». En: *Logic for Programming and Automated Reasoning*. Ed. por Michel Parigot y Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, págs. 85-95 (vid. pág. 30).

[15] R. Fervari. «Relation-Changing Modal Logics». Tesis doct. Facultad de Matemática, Astronomía y Física, Universidad Nacional de Córdoba, 2014 (vid. págs. 5, 17).

[16] Georges Gonthier. «Formal Proof — The Four-Color Theorem». En: 2008 (vid. pág. 4).

[17] Thomas C. Hales y col. «A formal proof of the Kepler conjecture». En: *CoRR* abs/1501.02155 (2015) (vid. pág. 4).

[18] J. Hintikka. *Knowledge and Belief. An Introduction to the Logic of the Two Notions*. Ithaca, NY: Cornell University Press, 1962 (vid. pág. 5).

[19] William A Howard. «The formulae-as-types notion of construction». En: *To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44 (1980), págs. 479-490 (vid. págs. 5, 30).

[20] Jacques-Henri Jourdan. «Verasco: a formally verified C static analyzer». Tesis doct. Universite Paris Diderot-Paris VII, 2016 (vid. pág. 4).

[21] Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas y Derek Dreyer. «Mtac2: Typed Tactics for Backward Reasoning in Coq». En: *Proc. ACM Program. Lang.* 2.ICFP (jul. de 2018), 78:1-78:31. URL: http://doi.acm.org/10.1145/3236773 (vid. pág. 30).

[22] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish y col. «seL4: Formal verification of an OS kernel». En: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM. 2009, págs. 207-220 (vid. pág. 4).

[23] Xavier Leroy y col. «The CompCert verified compiler». En: *Documentation and user's manual. INRIA Paris-Rocquencourt* 53 (2012) (vid. pág. 4).

[24] C. Lewis. *A Survey of Symbolic Logic*. Republished by Dover, 1960. University of California Press, 1918 (vid. pág. 5).

[25] Catherine A Meadows. «Formal verification of cryptographic protocols: A survey». En: *International Conference on the Theory and Application of Cryptology*. Springer. 1994, págs. 133-150 (vid. pág. 4).

[26] Norman D. Megill. «Metamath». En: *The Seventeen Provers of the World, Foreword by Dana S. Scott*. 2006, págs. 88-95. DOI: 10.1007/11542384_13. URL: https://doi.org/10.1007/11542384_13 (vid. pág. 4).

[27] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn y Jakob von Raumer. «The Lean Theorem Prover (System Description)». En: *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*. 2015, págs. 378-388. DOI: 10.1007/978-3-319-21401-6_26. URL: https://doi.org/10.1007/978-3-319-21401-6_26 (vid. pág. 4).

[28] Lawrence C Paulson. *Isabelle: A generic theorem prover*. Vol. 828. Springer Science & Business Media, 1994 (vid. pág. 4).

[29] Lawrence C. Paulson. «The Relative Consistency of the Axiom of Choice - Mechanized Using Isabelle/ZF». En: *CiE*. 2008 (vid. pág. 4).

[30] A. Prior. *Time and Modality*. Oxford University Press, 1957 (vid. pág. 5).

[31] Henrik Sahlqvist. «Completeness and correspondence in the first and second order semantics for modal logic». En: *Studies in Logic and the Foundations of Mathematics*. Vol. 82. Elsevier, 1975, págs. 110-143 (vid. pág. 66).

[32] Konrad Slind y Michael Norrish. «A brief overview of HOL4». En: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2008, págs. 28-32 (vid. pág. 4).

[33] Paulien de Wind. *Modal Logic in Coq*. Vrije Universiteit, 2001 (vid. págs. 3, 65, 66).

[34] G. H. von Wright. *An Essay in Modal Logic*. Amsterdam, North-Holland Pub. Co., 1951 (vid. pág. 5).

[35] Bruno Xavier, Carlos Olarte, Giselle Reis y Vivek Nigam. «Mechanizing Focused Linear Logic in Coq». En: *Electr. Notes Theor. Comput. Sci.* 338 (2018), págs. 219-236 (vid. pág. 66).