Universidad Nacional de Rosario

Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Computational Tree Logic Interpreter

Análisis de Lenguajes de Programación - R322

Sferco, Martín S-5656/1

23 de Febrero, 2026

Índice

1. Computer Tree Logic

1.1. Introducción a la verificación de modelos

La *verificación de modelos* es una técnica automática utilizada para verificar si un determinado sistema cumple con una especificación dada. Dado que estos sistemas suelen presentar comportamientos infinitos, la verificación formal requiere de tres componentes esenciales:

- **Un lenguaje de descripción de sistemas:** Generalmente sistemas de transiciones finitos.
- **Un lenguaje de especificación:** Lógicas temporales que permitan expresar propiedades que varían con el tiempo.
- **Un mecanismo de verificación:** Un algoritmo que determine automáticamente si el modelo $\mathcal{M}$ satisface la fórmula ϕ.

En nuestro caso, para el lenguaje de especificación, utilizaremos **Computer Tree Logic (CTL)**. Dicha lógica es fundamental en la verificación de modelos, y viene a complementar a otras lógicas como LTL (Linear Temporal Logic). A diferencia de las lógicas lineales, CTL permite razonar sobre las múltiples bifurcaciones que puede tomar la ejecución de un programa.

1.2. Sintaxis CTL

La sintaxis de CTL se construye a partir de proposiciones atómicas, conectivos lógicos, cuantificadores sobre caminos y operadores temporales. El conjunto de fórmulas CTL válidas se define como el menor conjunto inductivo que satisface las siguientes reglas:

- $\bot \in CTL$ y $p_i \in CTL$ para toda proposición atómica
- Si $\phi \in CTL$, entonces $(\neg\phi) \in CTL$
- Si $\phi, \psi \in CTL$, entonces $(\phi \wedge \psi) \in CTL$
- Si $\phi, \psi \in CTL$, entonces los siguientes son fórmulas: $\forall \bigcirc \phi \in CTL$ (ForAll Next)
- Si $\phi, \psi \in CTL$, entonces los siguientes son fórmulas: $\exists \bigcirc \phi \in CTL$ (Exists Next)
- Si $\phi, \psi \in CTL$, entonces los siguientes son fórmulas: $\forall[\phi \,\mathcal{U}\, \psi] \in CTL$ (Forall Until)
- Si $\phi, \psi \in CTL$, entonces los siguientes son fórmulas: $\exists[\phi \,\mathcal{U}\, \psi] \in CTL$ (Exist Until)

Hay otra forma de definir la sintaxis, utilizando dos definiciones de gramáticas las cuales son mutuamente recursivas: una para cuantificadores de caminos, y otra para predicados de estados. El resultado final es el mismo, pero nos pareció más adecuada la manera en la que lo presentamos.

Para definir $\phi \to \psi$, $\top$, y $\phi \vee \psi$, usamos las conocidas equivalencias lógicas con los conectivos lógicos que tenemos en nuestra sintaxis. Además, para ampliar un poco la riqueza de la sintaxis de nuestra lógica, y evitar formulas gigantes, se definen algunos operadores derivados:

- $\forall \diamond \phi \equiv \forall[\top \,\mathcal{U}\, \phi]$ (Inevitable)
- $\exists \diamond \phi \equiv \exists[\top \,\mathcal{U}\, \phi]$ (Possible)
- $\forall \Box \phi \equiv \neg\exists \diamond \neg\phi$ (Invariant)
- $\exists \Box \phi \equiv \neg\forall \diamond \neg\phi$ (Invariant for some trace)

1.3. Sistemas de transiciones

Antes de presentar la semántica de las diferentes fórmulas de la lógica, necesitamos definir lo que utilizaremos para describir nuestros distintos sistemas. Un **Sistema de Transiciones** $\mathcal{M}$ se define como una tupla $(S, \rightarrow, I, L)$ donde:

- S es un conjunto finito de **estados**.
- $I \subseteq S$ es el conjunto de **estados iniciales**.
- $\rightarrow \subseteq S \times S$ es una **relación de transición** no bloqueante (todo estado tiene al menos un sucesor).
- $L : S \rightarrow \mathcal{P}(AT)$ es una **función de etiquetado** que asigna proposiciones atómicas a cada estado.

El conjunto AT es cualquier conjunto no vacío conformado por proposiciones atómicas. Notemos además que, en nuestro caso, nos quedaremos con sistemas de transiciones con un número finito de estados.

Para representar *computaciones* o *ejecuciones*, utilizaremos el concepto de **traza**. Una traza es una secuencia infinita de estados

$$s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \cdots$$

tal que se cumple que

$$\forall i \in \mathbb{N}_0 : (s_i, s_{i+1}) \in \rightarrow$$

.

1.4. Semántica CTL

Ya habiendo definido la sintaxis de nuestra lógica y cómo vamos a representar nuestros sistemas de transiciones, podemos definir formalmente la semántica de una fórmula. La semántica se define como la relación de satisfacción

$$\mathcal{M}, s \models \phi$$

que se entiende como que la fórmula ϕ es válida en el sistema $\mathcal{M}$ si comenzamos en el estado s. Dicha relación se define por inducción sobre la estructura de la fórmula:

- $\mathcal{M}, s \models p_i \iff p_i \in L(s)$.
- $\mathcal{M}, s \models \neg\phi \iff \mathcal{M}, s \not\models \phi$.
- $\mathcal{M}, s \models \phi \wedge \psi \iff \mathcal{M}, s \models \phi$ y $\mathcal{M}, s \models \psi$.
- $\mathcal{M}, s \models \exists \bigcirc \phi \iff \exists s' \in S$ tal que $s \rightarrow s'$ y $\mathcal{M}, s' \models \phi$.
- $\mathcal{M}, s \models \forall \bigcirc \phi \iff \forall s' \in S$ tal que $s \rightarrow s'$ vale que $\mathcal{M}, s' \models \phi$.
- $\mathcal{M}, s \models \exists[\phi \, \mathbf{U} \, \psi] \iff$ existe una traza $s_0 \rightarrow s_1 \rightarrow \ldots$ con $s = s_0$ tal que $\exists j \in \mathbb{N}_0$ donde $\mathcal{M}, s_j \models \psi$ y $\forall i < j, \mathcal{M}, s_i \models \phi$.
- $\mathcal{M}, s \models \forall[\phi \, \mathbf{U} \, \psi] \iff$ para toda traza $s_0 \rightarrow s_1 \rightarrow \ldots$ tal que $\exists j \in \mathbb{N}_0$ donde $\mathcal{M}, s_j \models \psi$ y $\forall i < j, \mathcal{M}, s_i \models \phi$.

También podemos, con un poco de desarrollo, derivar la semántica de los operadores derivados que presentamos anteriormente:

- $\mathcal{M}, s \models \forall \diamond \phi \iff$ para toda traza $s_0 \rightarrow s_1 \rightarrow \ldots$ con $s = s_0, \exists j \in \mathbb{N}_0$ tal que $\mathcal{M}, s \models s_j$.

- $\mathcal{M}, s \models \exists \diamond \phi \iff$ para alguna traza $s_0 \to s_1 \to \ldots$ con $s = s_0, \exists j \in \mathbb{N}_0$ tal que $\mathcal{M}, s \models s_j$.
- $\mathcal{M}, s \models \forall \Box \phi \iff$ para toda traza $s_0 \to s_1 \to \ldots$ con $s = s_0, \mathcal{M}, s \models s_j, \forall j \in \mathbb{N}_0$.
- $\mathcal{M}, s \models \exists \Box \phi \iff$ para alguna traza $s_0 \to s_1 \to \ldots$ con $s = s_0, \mathcal{M}, s \models s_j, \forall j \in \mathbb{N}_0$.

Ya teniendo bien definida la relación de satisfacción para un determinado sistema de transiciones y un estado, podemos ampliar la definición. Decimos que un sistema de transiciones *modela* una fórmula, y lo indicamos como $\mathcal{M} \models \phi$, cuando se tiene que

$$\forall s \in I, \qquad \mathcal{M}, s \models \phi$$

es decir, la fórmula es válida en todos los estados iniciales del sistema. Por otro lado, decimos que una fórmula es *satisfacible*, y lo indicamos como $\models \phi$, si existe algún sistema $\mathcal{M}$ tal que $\mathcal{M} \models \phi$.

2. Algoritmo de Satisfactibilidad para CTL

Como hablamos en la sección de introducción, luego de que tenemos una forma de describir sistemas (nuestros sistemas de transición) y de que definimos un lenguaje de especificación (en nuestro caso, CTL), tenemos que pasar a definir un mecanismo de verificación automática que nos permita determinar si un determinado sistema de transiciones modela nuestra fórmula de especificación (también se puede pensar que nuestro sistema cumple los requerimientos de nuestra fórmula).

Para ésto, dada una fórmula ϕ, definimos el siguiente conjunto:

$$\text{Sat}(\phi) = \{s \in S : \mathcal{M}, s \models \phi\}$$

que representa todos los estados del sistema que verifican nuestra fórmula. Finalmente, simplemente debemos verificar si $I \subseteq \text{Sat}(\phi)$.

En el resto de la sección aprenderemos cómo se calcula dicho conjunto Sat. Dejaremos pendiente para la implementación del lenguaje analizar cómo se realiza la búsqueda de contra-ejemplos para los casos en los que el sistema no modele la fórmula.

2.1. Transformación a fórmulas equivalentes

Para facilitar el cálculo del conjunto $\text{Sat}(\phi)$, transformaremos a ϕ en una fórmula equivalente utilizando solo un conjunto mínimo de operadores. De esta manera, tendremos que definir un número menor de procedimientos que calculen nuestro conjunto de estados que satisfacen la fórmula. Definimos la función de transformación $T : CTL \to CTL$ de la siguiente manera:

$$T(p_i) = p_i \tag{1}$$
$$T(\bot) = \bot \tag{2}$$
$$T(\top) = \neg\bot \tag{3}$$
$$T(\neg\phi) = \neg T(\phi) \tag{4}$$
$$T(\phi \wedge \psi) = T(\phi) \wedge T(\psi) \tag{5}$$
$$T(\phi \vee \psi) = \neg(\neg T(\phi) \wedge \neg T(\psi)) \tag{6}$$
$$T(\phi \to \psi) = \neg(T(\phi) \wedge \neg T(\psi)) \tag{7}$$
$$T(\exists \bigcirc \phi) = \exists \bigcirc T(\phi) \tag{8}$$
$$T(\forall \bigcirc \phi) = \forall \bigcirc T(\phi) \tag{9}$$
$$T(\exists[\phi \ \mathbf{U} \ \psi]) = \exists[T(\phi) \ \mathbf{U} \ T(\psi)] \tag{10}$$
$$T(\forall[\phi \ \mathbf{U} \ \psi]) = \neg\exists[\neg T(\psi) \ \mathbf{U} \ (\neg T(\phi) \wedge \neg T(\psi))] \wedge \forall \diamond T(\psi) \tag{11}$$
$$T(\forall \diamond \phi) = \forall \diamond T(\phi) \tag{12}$$
$$T(\exists \diamond \phi) = \exists[\neg\bot \ \mathbf{U} \ T(\phi)] \tag{13}$$
$$T(\forall\Box\phi) = \neg\exists[\neg\bot \ \mathbf{U} \ \neg T(\phi)] \tag{14}$$
$$T(\exists\Box\phi) = \neg\forall \diamond \neg T(\phi) \tag{15}$$
$$\tag{16}$$

2.2. Conjuntos especiales de estados

Ahora, definiremos algunos procedimientos y conjuntos de estados que nos serán útiles para definir el conjunto Sat.

$$\text{pre}_{\exists}(Y) = \{s \in S : \exists s', s \to s' \text{ y } s' \in Y\}$$

$$\text{pre}_{\forall}(Y) = \{s \in S : \forall s', s \to s' \implies s' \in Y\}$$

En términos intuitivos:

- Un estado pertenece a $\text{pre}_{\exists}(Y)$ si tiene **algún** sucesor en Y.
- Un estado pertenece a $\text{pre}_{\forall}(Y)$ si tiene a **todos** sus sucesores en Y.

2.3. Procedimientos

El cálculo de la Sat para fórmulas complejas como $\forall \diamond \psi$ y $\exists[\phi \; \mathcal{U} \; \psi]$ se basa en alcanzar un punto fijo mediante iteraciones sucesivas. Dichos procedimientos se aplican sobre los conjuntos Sat de las subfórmulas de la fórmula correspondiente.

Procedimiento inev: Este procedimiento ayuda a calcular el conjunto $\text{Sat}(\forall \diamond \psi)$, identificando los estados donde la propiedad ψ es inevitable.

Listing 1: Algoritmo de Inevitabilidad

```
1  inev(Y) {
2      while (Y != Y U pre_forall(Y)) do
3          Y = Y U pre_forall(Y);
4      return Y;
5  }
```

En esencia, estamos yendo hacia atrás y únicamente agregamos los nodos para los cuales todos sus vecinos caen en el conjunto Y, el cual vamos ampliando iteración a iteración.

Procedimiento exuntil: Este procedimiento ayuda a calcular $\text{Sat}(\exists[\phi \; \mathcal{U} \; \psi])$, encontrando los estados desde los cuales existe un camino que satisface ϕ hasta llegar a un estado que cumple ψ.

Listing 2: Algoritmo de Existe Until

```
1  exuntil(X, Y) {
2      while (Y != Y U (X intersect pre_exists(Y))) do
3          Y = Y U (X intersect pre_exists(Y));
4      return Y;
5  }
```

Básicamente, estamos yendo hacia atrás y únicamente agregamos los nodos para los cuales todos sus vecinos caen en el conjunto Y (que vamos ampliando iteración a iteración), y que además sean elementos de X.

2.4. Definición del conjunto Sat para fórmulas transformadas

Ahora finalmente podemos definir la función $\text{Sat} : \text{CTL} \to \mathcal{P}(S)$ por recursión sobre la estructura de las fórmulas transformadas. Esta función devuelve el conjunto de estados donde una fórmula es válida.

- $\text{Sat}(\bot) = \emptyset$.
- $\text{Sat}(p_i) = \{s \in S : p_i \in L(s)\}$.
- $\text{Sat}(\neg\psi) = S - \text{Sat}(\psi)$.
- $\text{Sat}(\phi \wedge \psi) = \text{Sat}(\phi) \cap \text{Sat}(\psi)$.
- $\text{Sat}(\exists \bigcirc \psi) = \text{pre}_{\exists}(\text{Sat}(\psi))$.
- $\text{Sat}(\forall \bigcirc \psi) = \text{pre}_{\forall}(\text{Sat}(\psi))$.
- $\text{Sat}(\forall \diamond \psi) = \text{inev}(\text{Sat}(\psi))$.
- $\text{Sat}(\exists[\phi \; \mathcal{U} \; \psi]) = \text{exuntil}(\text{Sat}(\phi), \text{Sat}(\psi))$.

2.5. Definición del conjunto Sat

Con los componentes anteriores definidos, el algoritmo para determinar el conjunto de satisfacción $\text{Sat}(\phi)$ para una fórmula arbitraria $\phi \in \text{CTL}$ se resume en un proceso de dos fases:

1. **Normalización:** Se aplica la función de transformación T a la fórmula original para obtener una fórmula equivalente $T(\phi)$ que utilice exclusivamente los operadores básicos $(\neg, \wedge, \exists\bigcirc, \forall\bigcirc, \exists\mathcal{U}, \forall\diamond)$.
2. **Evaluación Recursiva:** Se calcula el conjunto $\text{Sat}(T(\phi))$ mediante la aplicación exhaustiva y recursiva de las reglas de satisfacibilidad, utilizando los procedimientos `inev` y `exuntil` cuando sea necesario.

Finalmente, decimos que el sistema de transiciones $\mathcal{M}$ es un modelo para la especificación $(\mathcal{M} \models \phi)$ si y solo si el conjunto de estados iniciales es un subconjunto del conjunto de satisfacción calculado:

$$I \subseteq \text{Sat}(T(\phi))$$

3. El lenguaje CTL

La idea de crear este lenguaje surge al terminar de cursar Lógica en el segundo año de la carrera. La verificación de modelos nos había resultado particularmente interesante, y nos parecía una gran idea contar con un programa que pudiera resolver en segundos lo que a nosotros nos llevaba varios minutos (e incluso varios intentos, debido a la confusión entre las distintas reglas).

El objetivo principal de este lenguaje es servir como una herramienta que facilite los primeros pasos en el aprendizaje de la verificación de modelos, en particular cuando se trabaja con Computational Tree Logic (CTL). Está enfocado en simplificar la definición de fórmulas y de sistemas de transición —los cuales pueden ser visualizados posteriormente— para luego verificar si los modelos definidos satisfacen las especificaciones dadas.

3.1. Alcances del lenguaje

El lenguaje CTL posibilita las siguientes tareas:

- **Definición de funciones de etiquetado** para indicar cuáles son las proposiciones atómicas que valen en un determinado estado.
- **Definición de funciones de transición** para indicar hacia qué nodos se puede llegar partiendo de un estado dado.
- **Definición de modelos** a partir de las funciones de etiquetado y transición.
- **Definición de fórmulas** de lógica temporal que permiten describir propiedades de los sistemas.
- **Verificación de especificaciones**, determinando si un modelo satisface una fórmula dada.
- **Búsqueda de modelos** para una fórmula, explorando distintos modelos hasta encontrar uno que la satisfaga o hasta alcanzar un límite prefijado.
- **Exportación de modelos** en formato PDF para facilitar su visualización.
- **Uso de un modo interactivo** para facilitar la experimentación con el lenguaje.
- **Interpretación de resultados** mediante informes detallados de la evaluación.

3.2. Pequeño manual de uso

CTLI está implementado en Haskell y utiliza Graphviz para la generación de gráficos. Es por eso que tendremos que instalar algunas dependencias antes de poder utilizar el lenguaje:

```
# Instalación de Stack
curl -sSL https://get.haskellstack.org/ | sh

# Instalación Graphviz
sudo apt-get update && sudo apt-get install -y graphviz libgraphviz-dev
cabal update && cabal install graphviz --lib

# Configuración y compilación del proyecto
stack setup && stack build
```

Ahora, para la utilización del lenguaje, tenemos las siguientes opciones:

```
stack exec -- ctl -e test/test.ctl     # Evaluación normal
stack exec -- ctl -t test/test.ctl     # Verificación de tipos1
stack exec -- ctl -i [test/test.ctl]   # Modo interactivo
```

En el último modo, las definiciones de un archivo se cargan, pero las verificaciones y exportaciones deben ejecutarse manualmente. En el archivo `README.md` se tiene una descripción completa del lenguaje, su sintaxis, definición de modelos y ejemplos de uso.

3.3. Sintaxis del lenguaje

A continuación, mostraremos la sintaxis abstracta de nuestro lenguaje:

$$
\begin{aligned}
\text{program} &::= \text{sentence program} \mid \epsilon \\
\text{sentence} &::= \textbf{define}\ \text{varIdent} :: \text{type} = \text{expr} \\
&\mid \textbf{export}\ \text{expr}\ \textbf{as}\ \text{file} \\
&\mid \text{expr} \models \text{expr} \\
&\mid \text{expr}, \text{nodeIdent} \models \text{expr} \\
&\mid \models \text{expr}\ \textbf{as}\ \text{file} \\
\\
\text{type} &::= \textbf{Model} \mid \textbf{Labels} \mid \textbf{Nodes} \mid \textbf{Formula} \\
\text{expr} &:= \text{modelExpr} \mid \text{formulaExpr} \mid \text{nodesExpr} \mid \text{labelsExpr} \mid \text{varExpr} \\
\text{varExpr} &:= \text{varIdent} \\
\text{modelExpr} &:= \langle \text{expr}, \text{expr} \rangle \\
\text{formulaExpr} &:= \text{formula} \\
\text{nodesExpr} &:= \{\ \text{transitionList}\ \} \\
\text{labelsExpr} &:= \{\ \text{labelingList}\ \} \\
\\
\text{formula} &:= \bot \mid \top \\
&\mid \text{atomIdent} \mid \text{varIdent} \\
&\mid \neg\text{formula} \\
&\mid (\text{formula}) \\
&\mid \text{formula} \wedge \text{formula} \mid \text{formula} \vee \text{formula} \mid \text{formula} \rightarrow \text{formula} \\
&\mid \forall \bigcirc \text{formula} \mid \exists \bigcirc \text{formula} \\
&\mid \forall \diamond \text{formula} \mid \exists \diamond \text{formula} \\
&\mid \forall\Box\text{formula} \mid \exists\Box\text{formula} \\
&\mid \forall[\text{formula}\ \mathbf{U}\ \text{formula}] \mid \exists[\text{formula}\ \mathbf{U}\ \text{formula}] \\
\\
\text{transitionList} &:= \text{transition transitionList} \mid \epsilon \\
\text{transition} &:= \text{nodeIdent} \implies \text{setNode} \\
\text{labelingList} &:= \text{labeling listAtomIdent} \mid \epsilon \\
\text{labeling} &:= \text{nodeIdent} \impliedby \text{setAtom} \\
\text{setAtom} &:= \{\ \text{listAtom}\ \} \\
\text{listAtom} &:= \epsilon \mid \text{atomIdent listAtom'} \\
\text{listAtom'} &:= \epsilon \mid \text{, atomIdent listAtom'} \\
\text{setNode} &:= \{\ \text{listNode}\ \} \\
\text{listNode} &:= \epsilon \mid \text{nodeIdent listNode'} \\
\text{listNode'} &:= \epsilon \mid \text{, nodeIdent listNode'} \\
\text{varIdent, atomIdent, nodeIdent, file} &:= \text{string}
\end{aligned}
$$

3.4. Decisiones de diseño

Convenciones de identificadores Una de las primeras decisiones de diseño fue establecer convenciones sintácticas distintas para los diferentes tipos de identificadores del lenguaje, con el objetivo de reducir ambigüedades. Optamos por las siguientes reglas:

- Los **nodos** de los sistemas de transición deben comenzar con el carácter `_`.
- Las **proposiciones atómicas** deben comenzar con una letra minúscula.
- Las **variables** (nombres de fórmulas, modelos, etc.) deben comenzar con una letra mayúscula.

Esta convención sintáctica permite distinguir de forma rápida el rol de cada identificador dentro del lenguaje.

Definición y fusión de modelos En la definición de modelos, tanto para la función de etiquetado como para la relación de transición, se permite que un mismo nodo aparezca en múltiples entradas. En ese caso, todas las apariciones del nodo se consideran como una sola definición, y sus correspondientes valores del lado derecho se combinan. Es decir, las etiquetas se unen y los conjuntos de nodos alcanzables se agregan.

Separación entre valores y expresiones El lenguaje distingue entre **expresiones** y **valores**. Si bien desde un punto de vista teórico (por ejemplo, en el λ-cálculo) los valores son un subconjunto de las expresiones que no pueden seguir reduciéndose, se optó por una separación explícita. La razón principal es práctica: los resultados del *parser* producen expresiones que requieren un procesamiento adicional para ser evaluadas. Separar valores de expresiones permite evitar la reevaluación innecesaria cuando una definición ya ha sido reducida.

Igualaciones y variables En las definiciones del tipo

```
define X :: Type = A
```

el lado derecho A puede ser tanto una variable previamente definida como una sub-expresión de una fórmula que referencia a una variable. Ésto se debe a que tenemos un tipo de expresión que representa una variable, pero también tenemos que una fórmula puede contener variables (pero hicimos que vivan en distintos mundos). Optamos por definir que en dichos casos la trataremos como una expresión variable.

3.5. Construcción de la Mónada

En esta sección se detalla la implementación de la mónada sobre la cual se estructura el lenguaje. El diseño se fundamentó en el cumplimiento de los siguientes requisitos:

- Acceso a estado global (definiciones y último archivo cargado).
- Acceso a configuración de ejecución que no cambiaba (modo de ejecución).
- Manejo de errores.
- Posibilidad de IO.

Para cumplir con todas estas especificaciones, definimos una clase `MonadCTL` y una implementación concreta `CTL` construida a partir de transformadores de mónadas. Utilizar transformadores de mónadas

nos pareció la forma más prolija de ir agregando las distintas capas de funcionalidades que nuestra mónada necesitaba.

La clase `MonadCTL` la definimos de la siguiente manera:

```
class (MonadIO m, MonadState GState m, MonadError Error m, MonadReader Conf m)↩
    => MonadCTL m
```

Esta clase es *vacía*, es decir, no introduce operaciones nuevas. El único propósito de esta clase es imponer un conjunto de capacidades mínimas que toda mónada que quiera ser de la clase `MonadCTL` debe poder hacer todo lo que pedimos anteriormente:

- **MonadState GState**: permite acceder a un estado global que incluye todas las definiciones declaradas y el último archivo cargado.
- **MonadError Error**: permite lanzar errores que incluyen información de donde ocurrió el error.
- **MonadReader Conf**: permite leer de una configuración, que incluye el modo de ejecución del intérprete. Estos modos pueden ser los de *Evaluar, Verificar Tipos, e Interactivo.*
- **MonadIO**: para realizar IO.

Luego, sobre la clase `MonadCTL`, definimos una interfaz de operaciones que permite utilizar la mónada, dándonos todas las funcionalidades necesarias para implementar el lenguaje.

Finalmente, la mónada concreta que utilizamos la definimos como:

```
type CTL = ReaderT Conf (StateT GState (ExceptT Error IO))
```

y por la naturaleza de los distintos transformadores de mónadas que utilizamos, el tipo `CTL` verifica trivialmente ser una instancia de `MonadCTL`, ya que verifica todos los constraints exigidos.

3.6. Representación interna y evaluación de expresiones

Una decisión central en el diseño de nuestro intérprete fue la separación estricta entre **expresiones** y **valores**. Mientras que las expresiones representan la sintaxis de entrada, los valores son el resultado de un proceso de reducción que los prepara para ser utilizados por los evaluadores. Estos valores se almacenan en el estado global de la mónada, asociados al identificador que les dio origen.

El proceso de reducción de expresiones se desglosa de la siguiente manera:

- **Variables:** Se resuelven sustituyéndolas por el valor ya computado que se encuentra almacenado en el estado global.
- **Modelos:** Se transforman en una estructura `TSystem`, compuesta por un grafo de transiciones y una función de etiquetado. Durante esta etapa se realizan validaciones críticas, como verificar que el grafo no contenga estados bloqueantes y que las etiquetas no hagan referencia a nodos externos al modelo.
- **Etiquetas:** Se procesan las definiciones de los nodos para generar una función de mapeo que unifica todas las proposiciones atómicas asociadas a cada estado.
- **Transiciones:** De forma análoga a las etiquetas, se transforman las definiciones de vecindad en una función que asocia cada nodo con su conjunto de sucesores.

- **Fórmulas:** Se realiza una conversión de *fórmula superficial* a *fórmula interna*. En este paso se reemplazan las ocurrencias de variables por sus valores correspondientes, garantizando la consistencia mediante el chequeo de tipos previo.

Al evaluar una sentencia, el sistema invoca este motor de reducción para transformar las expresiones en datos accionables antes de ejecutar la lógica de validación o exportación.

3.7. El módulo `Sat.hs`

Este módulo constituye el núcleo algorítmico del proyecto. En él se implementan las funciones encargadas de la manipulación lógica y la verificación de modelos:

- **Cálculo de `Sat`:** Implementa la función que determina el conjunto de estados de un sistema de transición que satisface una fórmula dada. Ésta es la herramienta principal para verificar si un modelo cumple con una especificación.
- **Transformación de fórmulas:** Contiene la lógica para convertir fórmulas complejas en variantes semánticamente equivalentes que facilitan su evaluación.
- **Generación de Sistemas:** Incluye un generador que, partiendo de un conjunto de átomos, construye de manera exhaustiva todos los sistemas de transición posibles, permitiendo realizar pruebas de cobertura.

Finalmente, el módulo gestiona la generación de **testigos** (*witnesses*) y **contraejemplos**. A través de las funciones `witness` y `counterexample`, el sistema identifica las razones por las cuales una fórmula se satisface o se invalida en un estado particular. Aunque presentan ciertas limitaciones de diseño, estas herramientas proporcionan información valiosa para que el usuario comprenda el comportamiento del modelo frente a la lógica definida. Los ejemplos que construimos surgen de analizar los vecinos de un estado en particular, o de intentar construir ciclos o caminos que cumplan ciertas particularidades respecto al conjunto Sat de algunas de las subfórmulas de la fórmula que estamos analizando.

3.8. Un sencillo sistema de tipos

Aunque nuestro lenguaje no contenga reducción de expresiones como tal, como sí lo hacen lenguajes como el λ-cálculo o lenguajes con expresiones aritméticas o booleanas, nos pareció útil incluir un sistema de tipos sencillo para evitar la aparición de errores en tiempo de ejecución durante la evaluación de los programas.

El lenguaje cuenta con un sistema de tipos *básico*, compuesto únicamente por cuatro tipos:

`ModelTy`, `NodesTy`, `LabelsTy`, `FormulaTy`.

Estos tipos corresponden directamente a las cuatro clases fundamentales de valores que maneja el lenguaje (etiquetas, transiciones, modelos y fórmulas). De esta manera, cada expresión del lenguaje puede ser clasificada en una de estas categorías.

El chequeo de tipos se realiza de manera estática, recorriendo las expresiones y sentencias del programa antes de su ejecución. Por ejemplo, se verifica que una fórmula solo utilice variables que hayan sido declaradas previamente como fórmulas.

El objetivo principal de este simple sistema es prevenir errores en *tiempos de ejecución*, como intentar evaluar una fórmula sobre un valor que no es un modelo. De igual manera, el lenguaje incluye verificaciones que se realizan sobre los valores durante la ejecución, en caso de que el chequeo de tipos haya fallado.

3.9. Un pequeño ejemplo:

```
// Definimos una funcion de etiquetado
define MazeLabels :: Labels = {
    _start <= { entrada }
    _room1 <= { p }
    _room2 <= { q }
    _room3 <= { q }
    _room3 <= { p }      // Lo mismo que _room3 <= { p, q }
    _trap  <= { dead }
    _exit  <= { win }
}

// Definimos una funcion de transicion
define MazeNodes :: Nodes = {
    (_start) => { _room1, _room2 } // _start es un nodo inicial
    _room1   => { _room3 }
    _room1   => { _trap }          // Lo mismo que _room1 => { _room3, _trap }
    _room2   => { _room1, _exit }
    _room3   => { _room2, _room3 }
    _trap    => { _trap }
    _exit    => { _exit }
}

// Definimos un modelo
define Maze :: Model = <MazeNodes, MazeLabels>

// Exportamos el modelo
export Maze as maze_check

// Evaluamos verificaciones
Maze |= E [] p               // Existe una traza en la que vale siempre p
Maze |= A <> win             // Todos los caminos te llevan a ganar
Maze, _trap |= A [] dead     // Desde la trampa, siempre vale que estamos ↩
    muertos.
```

3.10. Separación en archivos

A continuación explicamos brevemente la organización del proyecto:

- **app/**
  - **main.hs**: Funcion principal del proyecto, junto con la definición del modo interactivo.
- **test/:** Archivos de ejemplos.
- **export/:** Carpeta donde se exportan los modelos.
- **src/**
  - **Common.hs**: Sinónimos de tipos generales y utilidades.

- **Error.hs**: Definición de errores.
- **Eval.hs**: Evaluador de sentencias y reducción de expresiones a valores.
- **EvalResult.hs**: Definición de resultados de evaluación.
- **Global.hs**: Definiciones de tipos de datos que componen a la mónada.
- **Lang.hs**: Definición del AST del lenguaje.
- **MonadCTL.hs**: Define funciones para mostrar expresiones del lenguaje y resultados de evaluación.
- **Parser.hs** Definición del parser.
- **Sat.hs**: Implementa los distintos algoritmos de SAT, además de la búsqueda de testigos y contraejemplos.
- **MonadCTL.hs**: Definición de la mónada y toda su interfaz.
- **TypeCheck.hs**: Verificador de tipos.
- **Model/**
  - **TSystem.hs**: Definición de sistemas de transición y grafos.
  - **TSystemMethods.hs**: Funcionalidades de los sistemas de transición y los grafos.

4. Trabajo futuro

A medida que fuimos realizando el trabajo práctico, surgieron varias ideas que no terminamos implementando, ya que consideramos que se iban fuera del alcance de la materia. Algunas de ellas fueron las siguientes:

- **Explicación del algoritmo SAT:** Incluir otra sentencia del lenguaje que permita generar un paso a paso de cómo se construye el conjunto Sat a partir de los llamados recursivos en las subfórmulas.
- **Generación total de testigos y contraejemplos:** Mejorar la implementación actual para encontrar testigos y contraejemplos para los cuantificadores universales y existenciales respectivamente. Para poder implementar ésto, tendríamos que comenzar a trabajar con caracterización de caminos, ya que se tendría que demostrar que una fórmula o su negación, vale para cualquier traza elegida.
- **Mejora en la construcción de modelos:** Mejorar los algoritmos de búsqueda de modelos para fórmulas específicas, incorporando cotas superiores de exploración que garanticen la terminación del algoritmo en problemas de alta complejidad. Por otro lado, para fórmulas complejas, optimizar el algoritmo para que utilice alguna clase de heurística al buscar sobre el espacio de posibilidades.
- **Mejora en la búsqueda de ciclos y caminos:** Optimizar los procedimientos de detección de ciclos y exploración de caminos.
- **Unificación de diferentes lógicas temporales:** Extender la construcción del intérprete para permitir el uso en conjunto o alternado de diferentes lógicas, tales como LTL, CTL* o Stochastic CTL, brindando al usuario la flexibilidad de elegir el formalismo que mejor se adapte a su especificación.