DSL para Lógica Modal

Bautista José Peirone

2024

Análisis de Lenguajes de Programación

1 Introducción

La lógica modal es una teoría formal para realizar deducciones, como en cualquier lógica, la cual se sostiene sobre una semántica basada en modelos. Esta lógica agrega dos operadores adicionales a los presentes en la lógica proposicional, $\Box$, $\Diamond$, que pueden ser interpretadas de varias formas, lo cual proveen de una gran utilidad a las familias de lógicas modales. Este lenguaje de dominio específico está pensado para llevar a cabo la evaluación semántica de formulas en este sistema.

La semántica en está lógica depende de una estructura conocido como Kripke Frame. Estas consisten de un grafo que representa estados y transiciones entre ellos, y una función de etiquetado que indica que proposiciones atómicas valen en cada estado.

2 Sintaxis del lenguaje

programa	::= *sentencia programa*
	| ϵ
sentencia	::= `"def"` *identificador* `"="` *formula*
	| `"set" "frame" "="` *grafo*
	| `"set" "tag" "="` *funcion*
	| *expresion*
expresion	::= `"assume"` *logica*
	| `"isSatis"` *formula*
	| `"isValid"` *formula*
	| *estado* `" ⊩ "` *formula*
logica	::= *identificadorLogica*
	| `"{"` *secuenciaAxiomas* `"}"`
secuenciaAxiomas	::= *secuenciaAxiomas* `","` *identificadorAxioma*
	| *identificadorAxioma*
	| ϵ
formula	::= *atomo*
	| *identificador*
	| $\star$ *formula*
	| *formula* $\oplus$ *formula*
	| *formula* `"["` *formula* `"/"` *atomo* `"]"`
	| `"("` *formula* `")"`
$\star$	::= `" ∼ "`
	| `"[]"`
	| `"<>"`
$\oplus$	::= `"and"` | `"&&"`
	| `"or"` | `"||"`
	| `"->"`
	| `"<->"`
identificadorLogica	::= `"Logic-"`N, *N nombre de lógica standard*
identificadorAxioma	::= `"Axiom-"`A, *A nombre de axioma standard*

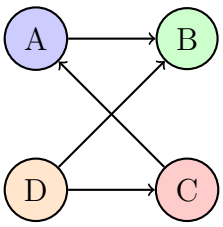
Los identificadores de formulas, es decir, los identificadores que pueden ser usados para referir a definiciones de formulas, son cadenas de caracteres alfanuméricos que inician en una letra

mayúscula. Por otro lado, los átomos son cadenas alfanuméricas que inician en caracteres en minúscula.

Por último, detallo la sintaxis para definir una función, como la de etiquetado. Esta misma sintaxis se reusa para la definición de un grafo, ya que los grafos, dirigidos en nuestro caso, pueden pensarse como una función que a cada vértice v le asigna el conjunto de vértices $\{w \in V | \ (v, w) \in E\}$. La gramatica para funciones es

$$
\begin{aligned}
funcion &::= \texttt{"\{"}\ secuenciaAsignaciones\ \texttt{"\}"} \\
secuenciaAsignaciones &::= secuenciaAsignaciones\ \texttt{","}\ asignacion \\
&\quad | \ asignacion \\
&\quad | \ \epsilon \\
asignacion &::= identificador\ \texttt{"->"}\ \texttt{"\{"}\ secuenciaIdentificadores\ \texttt{"\}"}
\end{aligned}
$$

El siguiente es un ejemplo de como puede especificarse un grafo arbitrario.




```
set frame = {
    A -> {B}
    C -> {A}
    D -> {B, C}
}
```

Como se ve, aquellos vértices que no tienen aristas salientes no tienen por qué ser incluidos en la especificación, el lenguaje infiere automáticamente que están en el grafo.

3 Semántica de la lógica modal

A continuación se presenta la definición de la relación del secuente utilizada en la lógica modal. Estas reglas son adaptadas al lenguaje, y cuando la relación vale, la expresión correspondiente retorna `True`, y cuando no lo hace, se retorna `False`. Sea un modelo $M = (V, E, T)$, entonces

$$
\begin{aligned}
w \Vdash p &\iff p \in T(w) \\
w \Vdash \phi \wedge \psi &\iff w \Vdash \phi \text{ y } w \Vdash \psi, \\
w \Vdash \phi \vee \psi &\iff w \Vdash \phi \text{ o } w \Vdash \psi, \\
w \Vdash \neg\phi &\iff w \nVdash \phi, \\
w \Vdash \phi \to \psi &\iff w \Vdash \phi \implies w \Vdash \psi. \\
w \Vdash \phi \leftrightarrow \psi &\iff w \Vdash \phi \iff w \Vdash \psi. \\
w \Vdash \Box\phi &\iff \forall v \in W, \text{ si } (w, v) \in E, \text{ entonces } v \Vdash \phi, \\
w \Vdash \Diamond\phi &\iff \exists v \in W \text{ tal que } (w, v) \in E \text{ y } v \Vdash \phi.
\end{aligned}
$$

Las operaciones de satisfacibilidad y validez se construyen sobre la relación del secuente, llamadas respectivamente `isSatis` y `isValid`, y definidas como

$$\frac{\forall w \in W,\ w \Vdash \phi}{\texttt{isValid}\ \phi} \qquad\qquad \frac{\exists w \in W,\ w \Vdash \phi}{\texttt{isSatis}\ \phi}$$

Por otro lado, las sentencias de asignación tienen una semántica más fácil de describir y no es necesario expresarlas con reglas. Simplemente definen aliases para fórmulas o configuran el modelo sobre el cual trabaja el programa.

4 Guía de Uso

El programa está desarrollado con la herramienta Stack de Haskell. Para compilar el programa basta con `stack build` y para correr el programa se utiliza `stack exec MLL-exe`.

El lenguaje soporta tres tipos de sentencias

1. Definiciones:

```
def P = p and q -> r
```

   Permite asignar un nombre a una formula. Esta formula luego puede ser utilizada mediante en posteriores expresiones, y es útil usar esta característica para definir estructuras comunes entre formulas y después utilizar reemplazos sintácticos sobre esta, en el caso del ejemplo, `P[s and t/p][s -> t/r]` sería equivalente a `(s and t) and q -> (s -> t)`. Los nombres de las definiciones deben empezar por mayúscula, y lo del los átomos por minúscula, para que no haya ambigüedades a la hora de evaluar las expresiones.

2. Modelos:

```
set frame = {
  x1 -> {x2,x3,x4},
  x1 -> {x5},
  x2 -> {x2},
  x3 -> {x1},
  x4 -> {x1, x2, x5},
  x6 -> {}
}

set tag = {
  x1 -> {},
  x2 -> {},
  x3 -> {p},
  x4 -> {p,q}
}
```

   Estos dos tipos de sentencias permiten modificar ambas partes de un modelo de manera independiente, por lo tanto se puede reutilizar un mismo frame con distintos tags, o el mismo tag con distintos frames. El formato concatena la parte derecha de todas las lineas que compartan lado izquierdo, por tanto aquí, en el frame del ejemplo, `x1` tiene aristas salientes a `x2, x3, x4 y x5`, ya que las lineas 2 y 3 se combinan. Por otro lado, no hace falta especificar el mapeo de cada vértice, aquellos que no estén presentes

se los toma como vértices sin aristas salientes. Como la sintaxis para especificar frames y tags es la misma, tal como se ve en el ejemplo, estas características se comparten para ambos tipos de estructuras.
Decidí asumir una hipótesis implícita del estilo Open World Assumption. Esto es, se pueden utilizar estados (osea, vértices del grafo) y átomos que no aparezcan en el modelo. Cuando se utiliza un estado que no ha sido especificado, se asume que no tiene aristas salientes y que ninguna proposición atómica vale en este. Me pareció interesante trabajar bajo esta hipótesis ya que permite de alguna forma trabajar sobre modelos infinitos de especificación finita.
En cualquier momento de la ejecución interactiva del programa, el comando `:frame` imprime el grafo y la función de etiquetado activas.

3. Expresiones:
Las expresiones se dividen en 4. Todas las expresiones ven modificada la forma en que muestran un resultado según sea indicado por una variable de estado conocida como `verbose`. Esta se modifica con el comando `:verbose` (o cualquier prefijo de la palabra), y cuando está activa, el output del programa es más extenso y muestra información sobre las trazas de evaluación de estas. Cuando este está desactivado, los resultados de las expresiones son simplemente respuestas booleanas.

   - Secuente

```
w ||- f
```

     Se evalúa el secuente de la formula f en el estado w. El secuente es la relación primitiva que define la semántica de toda la lógica modal. Esta operación, aunque toma dos argumentos de forma explicita, tiene un tercer argumento que es el modelo sobre el cual se evalúa. El modelo es implícito porque se utiliza siempre el modelo que esta activo al momento de evaluar la expresión.
     El modo verbose muestra toda la traza de evaluación, mostrando por qué cada operador proposicional y modal evalúa al resultado que tiene.

   - Validez y satisfacibilidad

```
isValid f
isSatis f
```

     Estas dos operaciones son parecidas, la primera verifica si todos los estados del modelo cumplen el secuente de la formula sobre ese estado, y la segunda solo chequea la existencia de algún estado que lo haga. Es decir, si el modelo es $M = (V, E, T)$, `isValid f` vale si y solo si $\forall$`w` $\in V$, `w ||- f`. Análogo para `isSatis f` cambiando el cuantificador universal por uno existencial.

Nombre	Fórmula	Propiedad en grafo
K	$\Box(\phi \to \psi) \to \Box\phi \to \Box\phi$	Vale en todos los grafos
T	$\Box\phi \to \phi$	Reflexividad
B	$\phi \to \Box\Diamond\phi$	Simetría
D	$\Box\phi \to \Diamond\phi$	Serial
4	$\Box\phi \to \Box\Box\phi$	Transitivdad
5	$\Diamond\phi \to \Box\Diamond\phi$	Euclideo
E	$\Box\phi \longleftrightarrow \Diamond\phi$	Funcional
C	$\Box(\phi \wedge \Box\phi \to \phi) \vee \Box(\phi \wedge \Box\phi \to \phi)$	Lineal

Table 1: Axiomas con sus definiciones y equivalencias en grafos

El modo verbose es equivalente al modo verbose del secuente, mostrando la traza sobre todos los estados del modelo.

- Verificación de lógicas
  Uno de los aspectos principales que tiene la lógica modal es la estrecha relación que existe entre algunas de sus formulas y propiedades sobre los grafos de los modelos. Resulta ser que, formulas como $\Box\phi \to \phi$ valen para cualquier formula modal ϕ en un modelo M si y solo si el grafo del M es un grafo reflexivo. Junto con esta, existen otras equivalencias que dejo listadas aquí 1. El acercamiento que toma el lenguaje es el siguiente. Dado un modelo ya cargado, la idea es poder preguntar cual de estos axiomas valen, ya que estos podrían ser requeridos para hacer alguna deducción o ser necesarios para la interpretación asignada al modelo. Es por esto que el lenguaje permite verificar cuales axiomas de los listados en la tabla valen analizando el grafo subyacente. El método más directo es con la sentencia

```
1 assume {Axiom-A1, Axiom-A2, ...}
```

  donde se indica el conjunto de los axiomas a validar por su nombre, prefijando la palabra 'Axiom' a cada uno. La segunda forma es indicándole el nombre de una lógica, que en esencia es un conjunto ya predeterminado de axiomas, las cuales muchas veces se usan de forma estándar y acarrean alguna interpretación propia de los axiomas u operadores, y por lo tanto el lenguaje provee las más comunes. Para utilizar la operación sobre una lógica estándar, se utiliza assume Logic−L L es el nombre de la lógica en cuestión.

  No se debe confundir las anteriores formulas con formulas normales. Los axiomas se piensan como esquemas sintácticos de fórmulas, cuando el axioma vale es porque esta vale para cualquier fórmula modal ϕ y ψ. Por eso, los axiomas abstraen propiedades generales que valen para cualquier formula. Un esquema así no puede ser probado por extensión sobre un modelo, ya que se deberían probar sustituyendo en todas las posibles formulas. Es por eso que las equivalencias en grafos nos proveen un método finito y computable para determinar la validez de

Lógica	Definición
K	$\{K\}$
T	$\{K, T\}$
S4	$\{K, T, 4\}$
KT45	$\{K, T, 4, 5\}$
S5	$\{K, T, B, 4, 5\}$
D	$\{K, D\}$

Table 2: Lógicas modales standard

dichas estructuras en el modelo.

En la siguiente tabla se ven las definiciones de cada lógica como conjuntos de axiomas. Una lógica es satisfecha si todos sus axiomas son válidos en el grafo.

Por último, si se usa el comando `:modal` se mostrara una lista con información similar a la presentada en estas tablas y con interpretaciones para cada lógica.

5 Notas de diseño

A continuación enumero algunas consideraciones o correcciones que hice a la hora de diseñar la sintaxis del lenguaje, intentando buscar un lenguaje más práctico de utilizar y menos propenso a que se cometan errores.

1. Uso de distintos espacios de nombres para identificadores y átomos. Los identificadores comienzan con mayúscula y los átomos con minúscula. La justificación detrás de esto es poder evitar sentencias confusas como `def p = r`, luego de la cual `p` queda registrado como una definición global; y un operador 'undef' no me parecía conveniente en el lenguaje. Por lo tanto, para salir de ambigüedades, las formulas globales (o lo que llamé esquemas) comienzan con mayúscula. Esto también me permite detectar errores de variables no definidas. Previo a este cambio, un identificador se buscaba como esquema, y si no existía es que se lo consideraba como átomo, por lo tanto nunca se podía afirmar que un identificador no estaba definido, sino que simplemente se pensaba como una proposición atómica mas.

2. El método para definir grafos en un principio contaba con una sentencia que definía sus vértices. Encontré que esto era impráctico, y es por eso que el algoritmo de construcción del grafo deduce de forma automática los vértices que participan en el grafo.

3. Con el fin de hacer un lenguaje mas practico y usable pensé en soportar varias notaciones para los operadores lógicos, basandome en la escritura que suelen tener los operadores lógicos en los lenguajes de programación, decidí usar `&&` y `||` como en C y `and` y `or` como Python en vez de utilizar algo sintacticamente más parecido en ASCII.

6 Referencias

- https://ncatlab.org/nlab/show/Kripke+frame
- http://intrologic.stanford.edu/dictionary/operator_precedence.html
- Logic in Computer Science de Michael Huth y Mark Ryan