Apunte de clase

{*log*}
Aplicaciones a la Especificación, Prototipado y Verificación de Software

Maximiliano Cristiá

Licenciatura en Ciencias de la Computación

Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Universidad Nacional de Rosario

Rosario – Argentina

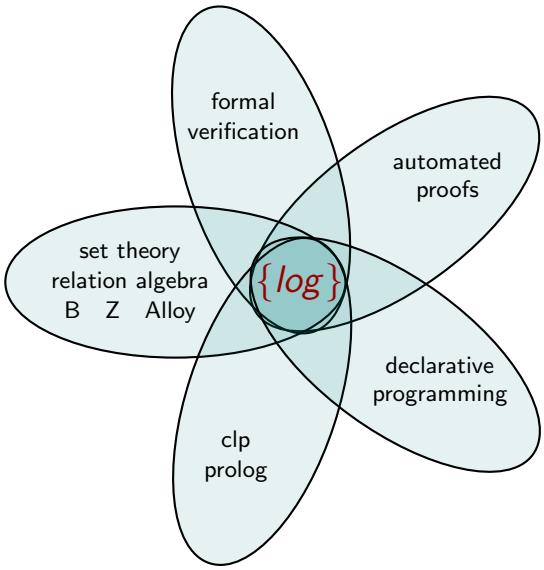
Índice

1. ¿Qué es {*log*}?

{*log*} ('setlog') es un lenguaje de programación basado en el paradigma de programación lógica de restricciones[1], pero además es un *satisfiability solver* y un demostrador automático de teoremas. Una de las características distintivas de {*log*} es que los conjuntos son entidades de primer nivel (es decir son parte integral del lenguaje).

El desarrollo de {*log*} lo comenzó Gianfranco Rossi en Italia a mediados de los años 90 junto a varios estudiantes de doctorado y colegas. Desde 2012 Gianfranco Rossi y Maximiliano Cristiá trabajan juntos en varias extensiones a {*log*} que permiten su aplicación a la especificación y verificación de software.

Como se puede ver en la figura de más bajo, {*log*} está en la intersección de varias áreas de las Ciencias de la Computación. {*log*} se puede usar como una herramienta para *verificación formal* porque es capaz de realizar *demostraciones automáticas* sobre una teoría de conjuntos muy expresiva. También es un *lenguaje de programación declarativo* lo que significa que los programadores pueden escribir la lógica del programa sin tener que describir el flujo de control. En particular, {*log*} implementa programación declarativa como una instancia de la *programación lógica de restricciones* (CLP) implementada en Prolog. Un programa {*log*} es muy similar (en su esencia, no en su forma) a especificaciones formales escritas en lenguajes basados en teoría de conjuntos y álgebra de relaciones conjuntista tales como Alloy, B y Z.



Este apunte se focaliza en mostrar cómo traducir o implementar especificaciones Z usando el lenguaje de {*log*} y, luego, cómo efectuar simulaciones, demostraciones automáticas y generación de casos de prueba usando el entorno {*log*}. La presentación es práctica, informal y orientada a un usuario que desea aprender a usar la herramienta. Para presentaciones técnicas sobre {*log*} y la teoría sobre la cual se construye pueden consultar artículos académicos [10, 11, 3, 2, 5, 7, 9, 4, 6, 8].

[1] 'constraint logic programming', en inglés.

1.1. Instalación

{*log*} está implementado en Prolog. Por lo tanto para poder usar {*log*} primero hay que instalar un intérprete de Prolog. Por el momento {*log*} solo funciona sobre el intérprete SWI-Prolog (`http://www.swi-prolog.org`). Luego se debe poner en cualquier directorio el contenido del archivo zip que contiene el código de {*log*}. Ese zip así como también todo lo relacionado con {*log*} lo pueden encontrar en el sitio oficial:

https://www.clpset.unipr.it/setlog.Home.html

Asumimos que ya estudiaron el tutorial sobre {*log*}.

Recomendamos enfáticamente que a medida que leen este apunte lean las secciones indicadas del manual de usuario de {*log*}.

1.2. Uso de {*log*}

Como ya dijimos, {*log*} es un *satisfiability solver*. Esto significa que {*log*} es un programa que determina si una fórmula dada es o no satisfacible. Cuando accedemos a {*log*} nos presenta un *prompt*:

```
{log}=>
```

Ahora podemos pedirle a {*log*} que resuelva una fórmula. Por ejemplo:

```
{log}=> un({a,2},B,{X,2,c}).
```

El predicado atómico `un({a,2},B,{X,2,c})` corresponde a $\{a,2\} \cup B = \{X,2,c\}$, donde X y B son variables mientras que a y c son constantes. En {*log*} los nombres de variable empiezan con una letra mayúscula, y las constantes empiezan con una letra minúscula. Observen que terminamos la fórmula con un punto. Entonces, cuando pulsamos Enter, {*log*} va a tratar de encontrar valores para `B` y `X` que satisfagan la fórmula—por este motivo decimos que {*log*} es un *satisfiability solver*. De esta forma, {*log*} se pregunta, ¿hay valores para `B` y `X` que hagan la fórmula verdadera? {*log*} responde lo siguiente:

```
B = {c},
X = a

Another solution?  (y/n)
```

Como pueden ver, {*log*} produce una solución y pregunta si queremos ver otra solución. En este caso hay tres soluciones más:

```
B = {2,c},
X = a
```

```
Another solution?  (y/n)
B = {a,c},
X = a

Another solution?  (y/n)
B = {a,2,c},
X = a

Another solution?  (y/n)
no

{log}=>
```

Cuando no hay más soluciones o cuando no tipeamos 'y', {*log*} dice 'no' e imprime el *prompt* otra vez.

Probemos otro ejemplo.

```
{log}=> un({a,2},B,{X,2,c}) & c nin B.
```

El predicado atómico `c nin B` corresponde a $c \notin B$ y '`&`' corresponde a la conjunción lógica ($\wedge$). En este caso {*log*} responde `no`. ¿Por qué? Porque no hay valores para `B` y `X` que hagan la fórmula verdadera. Claramente, como `c` no pertenece a `{a,2}` pero al mismo tiempo pertenece a la unión entre ese conjunto y `B`, entonces la única posibilidad de satisfacer la fórmula es cuando `c` pertenece a `B`. Pero esa posibilidad se cancela debido a la restricción `c nin B`. Entonces, {*log*} nos está diciendo "tu fórmula no es satisfacible".

Resumiendo, si vemos algo distinto a 'no' sabemos que la fórmula es satisfacible; en cualquier otro caso, es insatisfacible.

1.3. Ayuda

{*log*} provee una ayuda en línea bastante completa que les puede resultar conveniente cuando estén resolviendo el TP.

```
{log}=> help.

Call h(+What) where What is:
  syn:  {log} syntax w.r.t. Prolog and syntax of set terms
  cons: {log} constraints
  q:    quantifiers provided by {log}
  type: {log} type system
  sm:   specification of state machines
  next: Next, a simple environment for running state machines
  opt:  execution options
  cmd:  {log} user commands
  lib:  {log} library predicates
  all:  to get all available help information
```

De esta forma, pueden ejecutar `h(syn)` para obtener ayuda sobre la sintaxis de {*log*}, `h(sm)` para obtener ayuda sobre la especificación de máquinas de estados en {*log*}, etc.

2. Implementar una especificación Z en {*log*}

Como la mayoría de las especificaciones Z tienen una estructura muy similar, comenzaremos mostrando cómo implementar o traducir una típica especificación Z a {*log*} por medio de un ejemplo. Luego explicaremos con cierto detalle cómo traducir los elementos de Z que no aparecen en el ejemplo o que se pueden traducir de más de una forma.

2.1. Un ejemplo de traducción

La especificación que usaremos como ejemplo es una de las especificaciones usadas por Spivey en varios de sus artículos y libros [12], conocida como la *agenda de cumpleaños* (*birthday book*, en inglés).

2.1.1. La especificación Z

Comenzamos dando algunas designaciones.

n es un nombre $\approx n \in NAME$

d es una fecha $\approx d \in DATE$

k es el nombre de una persona cuyo cumpleaños hay registrar $\approx k \in known$

La fecha de cumpleaños de la persona $k \approx birthday\ k$

Entonces introducimos los siguientes tipos básicos.

$[NAME, DATE]$

Ahora podemos definir el espacio de estados de la especificación de la siguiente forma.

BirthdayBook

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \pfun DATE$$

El siguiente esquema describe los predicados que deberían ser invariantes de estado.

BirthdayBookInv

$$BirthdayBook$$

$$known = \mathrm{dom}\, birthday$$

El estado inicial de la agenda de cumpleaños es el siguiente.

BirthdayBookInit

$$BirthdayBook$$

$$known = \emptyset$$
$$birthday = \emptyset$$

La primera operación que modelamos es cómo agregar una fecha de cumpleaños a la agenda. Como siempre modelamos primero el caso exitoso, luego los errores y finalmente integramos todo en una única expresión de esquemas.

AddBirthdayOk

$$\Delta BirthdayBook$$
$$name? : NAME$$
$$date? : DATE$$

$$name? \notin known$$
$$known' = known \cup \{name?\}$$
$$birthday' = birthday \cup \{name? \mapsto date?\}$$

NameAlreadyExists

$$\Xi BirthdayBook$$
$$name? : NAME$$

$$name? \in known$$

$$AddBirthday == AddBirthdayOk \lor NameAlreadyExists$$

La segunda operación a especificar corresponde a mostrar el cumpleaños de una persona dada.

FindBirthdayOk

$$\Xi BirthdayBook$$
$$name? : NAME$$
$$date! : DATE$$

$$name? \in known$$
$$date! = birthday(name?)$$

NotAFriend

$$\Xi BirthdayBook$$
$$name? : NAME$$

$$name? \notin known$$

$$FindBirthday == FindBirthdayOk \lor NotAFriend$$

Finalmente tenemos una operación que nos lista los nombres de las personas cuya fecha de cumpleaños es hoy.

Remind

$$\Xi BirthdayBook$$
$$today? : DATE$$
$$cards! : \mathbb{P}\, NAME$$

$$cards! = \mathrm{dom}(birthday \rhd \{today?\})$$

2.1.2. El código {*log*}

Esta sección la pueden completar leyendo la sección 13.1 del manual del usuario de {*log*}.

El código {*log*} de la traducción de la especificación Z se debe guardar en un archivo con extensión `pl` o `slog` preferentemente en el mismo directorio donde fue instalado {*log*}.

La mayoría de los esquemas Z se traducen a predicados {*log*}. Un predicado {*log*} es algo así como un procedimiento o subrutina. Cada predicado puede recibir cero o más parámetros. Cuando se traduce un esquema Z que representa una operación del sistema, los parámetros que recibe el predicado correspondiente serán las variables de estado, las de entrada y las de salida.

En {*log*} las variables siempre deben empezar con una letra mayúscula o el caracter de subrayado (_), aunque este en general queda reservado para casos especiales. Cualquier identificador que comienza con una letra minúscula es una constante.

En {*log*} el caracter prima (') no se puede usar como parte del nombre de una variable. Por lo tanto para identificar las variables de estado posterior pondremos el caracter de subrayado al final. Por lo tanto, por ejemplo, las variables de estado de la especificación de la agenda de cumpleaños serán `Known` y `Birthday`; y las de estado posterior `Known_` y `Birthday_`.

De igual forma, las decoraciones ? y ! no pueden formar parte de los nombres de variables {*log*}. En este caso no usaremos ninguna convención particular para distinguir entradas de salidas.

Las variables de estado se declaran de esta forma:

```
variables([Known,Birthday]).
```

Notar que la declaración termina en un punto. Más adelante veremos cómo declarar el tipo de cada variable.

El invariante de estado se escribe de la siguiente manera:

```
invariant(birthdayBookInv).
birthdayBookInv(Known,Birthday) :- dom(Birthday,Known).
```

O sea que `invariant(birthdayBookInv)` declara que el predicado `birthdayBookInv` es un invariante. Notar que si bien *BirthdayBookInv* comienza con mayúscula `birthdayBookInv` lo hace con minúscula porque los predicados {*log*} deben comenzar con minúscula. El predicado `dom(Birthday,Known)` se interpreta como dom *Birthday* = *Known*. En la sección 5 veremos más sobre la traducción de invariantes.

Los invariantes se declaran entre la declaración de las variables de estado y el estado inicial que se traduce así:

```
initial(birthdayBookInit).
birthdayBookInit(Known,Birthday) :-
  Known = {} &
  Birthday = {}.
```

donde `&` equivale a la conjunción y `{}` es la forma de escribir el conjunto vacío ($\emptyset$).

La interfaz del predicado {*log*} correspondiente al esquema Z *AddBirthdayOk* es la siguiente:

```
addBirthdayOk(Known,Birthday,Name,Date,Known_,Birthday_)
```

donde `Name` y `Date` corresponden a las variables de entrada *name*? y *date*? declaradas en *AddBirthday*. Como *name*? y *date*? son variables, en la cláusula las escribimos como `Name` y `Date`. Por otro lado, `Known` y `Birthday` representan el estado de partida mientras que `Known_` y `Birthday_` representan el estado de llegada. Ahora podemos dar la definición de la cláusula `addBirthdayOk`:

```
addBirthdayOk(Known,Birthday,Name,Date,Known_,Birthday_) :-
  Name nin Known &
  un(Known,{Name},Known_) &
  un(Birthday,{[Name,Date]},Birthday_).
```

donde `Name nin Known` significa $Name \notin Known$; `un(Known,{Name},Known_)` significa $Known\_ = Known \cup \{Name\}$; y `un(Birthday,{[Name,Date]},Birthday_)` significa $Birthday' = Birthday \cup \{(Name, Date)\}$.

Ahora veamos la definición de `nameAlreadyExists`:

```
nameAlreadyExists(Known,Birthday,Name,Known_,Birthday_) :-
  Name in Known &
  Known_ = Known &
  Birthday_ = Birthday.
```

donde podemos ver cómo se indica que no hay cambio de estado. Observar que a `nameAlreadyExists` no se le pasa el parámetro `Date` porque no se declara en *NameAlreadyExists*.

Finalmente podemos dar la traducción de *AddBirthday*:

```
operation(addBirthday).
addBirthday(Known,Birthday,Name,Date,Known_,Birthday_) :-
  addBirthdayOk(Known,Birthday,Name_Date,Known_,Birthday_)
  or
  nameAlreadyExists(Known,Birthday,Name,Known_,Birthday_).
```

donde `operation` declara que la cláusula `addBirthday` es una operación. Además, `or` equivale a la disyunción lógica. Notar que `addBirthdayOk` y `nameAlreadyExists` *no* están precedidas por una declaración `operation` porque ya forman parte de una operación.

Hasta el momento no hemos indicado los tipos de las variables. {*log*} es un formalismo esencialmente no tipado pero las últimas versiones incorporan un sistema de tipos similar al de

Z. El sistema de tipos de {*log*} está descripto en detalle en la sección 12 del manual del usuario de {*log*}. Aquí daremos los lineamientos generales del sistema de tipos; para más información consultar el manual de {*log*}.

El sistema de tipos de {*log*} permite definir sinónimos de tipos que pueden ayudar a simplificar el tipado de cláusulas y variables. Por ejemplo podemos definir los siguientes sinónimos:

```
def_type(bb,rel(name,date)).
def_type(kn,set(name)).
```

donde `bb` es un identificador o sinónimo del tipo `rel(name,date)`. En `rel(name,date)`, `name` y `date` corresponden a los tipos básicos *NAME* y *DATE* de la especificación Z. En {*log*} los tipos básicos de Z se pueden introducir sin ninguna declaración previa. En {*log*} los tipos básicos siempre deben empezar con una letra minúscula (es decir, son constantes). Además, `rel(name,date)` corresponde al tipo de las relaciones binarias entre `name` y `date` (i.e., $NAME \leftrightarrow DATE$ en Z). Por otro lado, `set(name)` corresponde al tipo de todos los conjuntos de tipo `name` (i.e. $\mathbb{P} NAME$ en Z).

Con estos sinónimos de tipos podemos declarar, por ejemplo, el tipo de la cláusula `addBirthday`:

```
dec_p_type(addBirthday(kn,bb,name,date,kn,bb)).
```

Esta declaración debe preceder a la definición del predicado:

```
operation(addBirthday).
dec_p_type(addBirthday(kn,bb,name,date,kn,bb)).
addBirthday(Known,Birthday,Name_i,Date_i,Known_,Birthday_) :-
  addBirthdayOk(Known,Birthday,Name_i,Date_i,Known_,Birthday_)
  or
  nameAlreadyExists(Known,Birthday,Name_i,Known_,Birthday_).
```

La idea es que el predicado `dec_p_type` tiene un único parámetro que es de la forma:

$$nombrePredicado(p_1, \ldots, p_n)$$

A su vez, cada p_i se corresponde uno a uno con los parámetros del predicado lo que permite declarar el tipo de cada uno. Entonces el tipo de `Known` es `kn`, el de `Birthday` es `bb`, etc.

En el Apéndice A pueden encontrar todo el código {*log*} de este ejemplo que incluye las declaraciones de tipos de todas las cláusulas que hemos visto.

Recordar que en Z las funciones parciales *no* son un tipo. Lo mismo ocurre en {*log*}; de hecho no es posible definir el tipo de las funciones parciales. Lo mismo ocurre con los números naturales. Esto significa que si en Z hemos declarado $f : X \pfun Y$ en {*log*} debemos declarar f con tipo `rel(x,y)` y luego demostrar que f es una función (veremos esto con más detalle más adelante). De forma similar, si en Z declaramos $x : \mathbb{N}$ en {*log*} debemos declarar x con tipo `int` y luego demostrar que vale `0 =< X`. En general, al traducir una especificación Z a {*log*} sería conveniente primero normalizar la especificación Z y luego hacer la traducción. En ese caso los tipos de Z se traducen directamente a {*log*} y los predicados introducidos a raíz de la normalización se introducen como restricciones (i.e. `0 =< X`) o se demuestra que son invariantes. Por ejemplo, $x : \mathbb{N}$ es una declaración no normalizada porque $\mathbb{N}$ no es un tipo (es un

conjunto). La declaración normalizada sería $x : \mathbb{Z}$ más $x \geq 0$ en la parte de predicados, en cuyo caso en {*log*} el tipo de x es `int` y deberíamos demostrar que x es siempre mayor o igual a cero.

En el ejemplo de la agenda de cumpleaños la declaración $birthday : NAME \pfun DATE$ no está normalizada. La declaración normalizada sería $birthday : NAME \rel DATE$ más $birthday \in NAME \pfun DATE$ en la parte de predicados. En este caso se traduce el tipo de *birthday* como hicimos más arriba y más adelante vamos a usar {*log*} para demostrar que *birthday* es una función. De hecho si hubiéramos escrito el esquema *BirthdayBook* con declaraciones normalizadas resultaría lo siguiente:

BirthdayBook

$known : \mathbb{P} NAME$
$birthday : NAME \rel DATE$

$birthday \in NAME \pfun DATE$

donde $birthday \in NAME \pfun DATE$ sería un invariante de estado por definición. Por otro lado, según vimos en Ingeniería de Software I [1], nosotros no usamos invariantes por definición sino que los escribimos en el esquema *BirthdayBookInv*. En consecuencia tendríamos:

BirthdayBook

$known : \mathbb{P} NAME$
$birthday : NAME \rel DATE$

BirthdayBookInv

BirthdayBook

$known = \mathrm{dom}\, birthday$
$birthday \in NAME \pfun DATE$

Entonces es claro que deberíamos traducir $birthday \in NAME \pfun DATE$ como un invariante. La traducción a {*log*} es la siguiente[2]:

```
invariant(birthdayPfun).
dec_p_type(birthdayPfun(bb)).
birthdayPfun(Birthday) :- pfun(Birthday).
```

`pfun` es un operador {*log*} que implementa la definición de función (parcial) como una subclase de los conjuntos de pares ordenados.

El segundo esquema de operación a traducir es *FindBirthday*. Como hicimos anteriormente, debemos comenzar por los esquemas que son invocados desde *FindBirthday*. Como *FindBirthday* no modifica el estado de la agenda podemos no mencionar las variables de estado de llegada.

```
dec_p_type(findBirthdayOk(kn,bb,name,date)).
findBirthdayOk(Known,Birthday,Name,Date) :-
```

[2]Recordar que los invariantes deben ir entre la declaración de variables y la declaración del estado inicial.

```
    Name in Known &
    applyTo(Birthday,Name,Date).

dec_p_type(notAFriend(kn,bb,name)).
notAFriend(Known,Birthday,Name) :- Name nin Known.
```

Observar que no hemos mencionado las variables `Known_` y `Birthday`. Además `Date` representa la variable Z *date*!; y `applyTo(Birthday,Name,Date)` significa *Birthday*(*Name*) = *Date*.

Ahora podemos ver la traducción de *FindBirthday*:

```
operation(findBirthday).
dec_p_type(findBirthday(kn,bb,name,date)).
findBirthday(Known,Birthday,Name,Date) :-
  findBirthdayOk(Known,Birthday,Name,Date)
  or
  notAFriend(Known,Birthday,Name,).
```

Notar que nuevamente usamos la declaración `operation`.

Finalmente la traducción de la operación *Remind* es la siguiente:

```
operation(remind).
dec_p_type(remind(kn,bb,date,kn)).
remind(Known,Birthday,Today,Cards) :-
  rres(Birthday,{Today},M) & dec(M,bb) &
  dom(M,Cards).
```

Este ejemplo es interesante porque podemos ver cómo se deben traducir las expresiones Z que involucran varios operadores de la teoría de conjuntos y relaciones. En efecto, como en {*log*} los operadores de la teoría de conjuntos se implementan como predicados, no es posible escribir expresiones. Lo que se debe hacer es introducir variables nuevas (como `M`) para "encadenar" los predicados. El predicado `rres(R,A,S)` equivale a $S = R \rhd A$. También es interesante porque nos permite introducir el predicado `dec(V,t)` cuya semántica es "la variable `V` es de tipo `t`".

3. Traducción de Z a {*log*}

Habiendo concluido con la traducción de la especificación de la agenda de cumpleaños, a continuación mostramos cómo implementar o traducir otros elementos de Z que no aparecieron en el ejemplo de la sección anterior.

3.1. Traducción de pares ordenados

Los pares ordenados se traducen a {*log*} como listas Prolog de dos elementos. Por ejemplo si x es una variable $(x, 3)$ se traduce como `[X,3]`.

Si en Z tenemos algo de la forma $p : \mathbb{Z} \times V$ y $p.1 = x - 4$ lo traducimos de la siguiente forma: `P = [A,_] & A is X - 4` (ver el operador `is` en la Sección 3.4). Es decir que, básicamente, usamos unificación para forzar que `P` sea una lista de dos elementos tal que el primero es la

variable `A` la cual a su vez se pide que sea igual a `X - 4`. `A` debe ser una variable no usada en la cláusula. Notar que usamos '_' porque no nos interesa el segundo parámetro de `P`.

Las listas Prolog no deben usarse para otras cosas que las que hemos indicado aquí. En general un programa {*log*} que usa listas Prolog de otras formas suele perder propiedades importantes.

3.2. Traducción de conjuntos

3.2.1. Conjuntos extensionales — Introducción a la unificación conjuntista

El conjunto {1,2,3} se traduce simplemente como `{1,2,3}`. Si uno de los elementos del conjunto es una variable o una constante de un tipo enumerado, hay que tener en cuenta las diferencias que hay entre Z y {*log*} en cuanto a constantes y variables. Por ejemplo, si en Z x es una variable, entonces el conjunto $\{2, x, 6\}$ se traduce como `{2,X,6}`.

{*log*} provee una forma de definir conjuntos por extensión que, en un sentido, es más poderosa que lo que ofrece Z. El término `{.../...}` se llama *constructor de conjuntos extensionales*. En `{`E`/`C`}` el segundo parámetro (i.e. C) debe ser un conjunto. La interpretación de `{`E`/`C`}` es $\{E\} \cup C$ por lo que es posible una solución donde $E \in C$. Si tal solución no es correcta o necesaria se debe conjugar el predicado $E \notin C$ de forma explícita; en general es conveniente agregar ese predicado. Para simplificar la entrada/salida, {*log*} acepta e imprime términos como `{1,2 / X}` en lugar de `{1 / {2 / X}}`.

El constructor extensional es muy útil y suele ser más eficiente que otras alternativas. Por ejemplo, el predicado Z:

$$A' = A \setminus \{d?\}$$

se puede traducir usando el predicado {*log*} `diff` cuya semántica es equivalente a \ (ver Tabla 1):

```
diff(A,{D},A_)
```

Pero también se puede traducir usando un conjunto extensional:

```
A = {D / A_} & D nin A_ or D nin A & A_ = A
```

lo que en general será más eficiente.

Es decir el predicado `A = {D / A_}` *unifica* `A` con `{D / A_}` de forma tal que encuentra valores para las variables que hagan que la igualdad sea verdadera. Si tales valores no existen, la unificación falla y {*log*} prueba con la segunda alternativa de la disyunción.

¿Por qué conjugamos `D nin A_`? Simplemente porque, por ejemplo, `A = {1,2}, D = 1` y `A_ = {1,2}` es una solución para la ecuación pero que no es una solución de $A' = A \setminus \{d?\}$. Precisamente, al conjugar `D nin A_` eliminamos todas las soluciones donde `D` pertenece a `A_`.

{*log*} resuelve las igualdades de la forma `B = C`, donde `B` y `C` son términos que denotan conjuntos, usando *unificación conjuntista*[3]. La unificación conjuntista está en la base del poder deductivo y de cómputo de {*log*} constituyendo una extensión importante del algoritmo de

[3] 'set unification', en inglés.

unificación sintáctica de Prolog. La unificación conjuntista es inherentemente computacionalmente pesada puesto que determinar si dos conjuntos son iguales o no implica, en el peor caso, calcular todas las permutaciones de sus elementos. A esto se agrega el hecho de que {*log*} unifica conjuntos *parcialmente especificados*, es decir conjuntos donde algunos de sus elementos o una parte de ellos son variables. Por este motivo, en general, {*log*} presentará problemas de eficiencia al intentar resolver ciertas fórmulas pero a la vez no sabemos de otras herramientas capaces de resolver ciertos problemas que {*log*} resuelve en tiempos razonables.

3.2.2. Producto cartesiano

Otra forma de construir un conjunto es mediante el producto cartesiano de dos conjuntos: `cp(A,B)`, donde A y B pueden ser cualesquiera conjuntos extensionales, es equivalente a $A \times B$.

3.2.3. Intervalos de números enteros

Un intervalo Z de la forma $n\,..\,m$ se traduce como `int(n,m)` teniendo en cuenta las diferencias entre variables y constantes que existen en {*log*}. Además los intervalos de {*log*} solo admiten constantes o variables como límites. Por lo tanto si en Z tenemos $x+1\,..\,2*h$ en {*log*} debemos escribir lo siguiente (ver la Sección 3.4 para la traducción de expresiones artiméticas):

```
int(A,B) & A is X + 1 & B is 2*H
```

3.3. Traducción de la aplicación de función

Una de las aplicaciones interesantes de la unificación conjuntista es la aplicación de una función a su argumento. Dado que en Z la mayor parte de las funciones que se usan en especificaciones son parciales es necesario agregar predidcados de la forma $x \in \mathrm{dom} f$, antes de aplicar f a x. La traducción a {*log*} de estas fórmulas puede hacerse de forma directa o usando convenientemente un predicado de pertenencia que da lugar a una unificación conjuntista. Por ejemplo, la fórmula Z:

$$x \in \mathrm{dom} f \land f\ x = y$$

puede traducirse de forma directa:

```
dom(F,D) & X in D & applyTo(F,X,Y)
```

o usando unificación conjuntista:

```
F = {[X,Y] / G} & [X,Y] nin G
```

donde en este caso se asume que `F` es una función (en general esta hipótesis se verifica cuando se prueba que `pfun(F)` es un invariante). Ciertamente, si en `F` no hay ningún par ordenado cuya primera componente sea `X` la unificación fallará lo que es equivalente a que $x \in \mathrm{dom} f$ sea falso. De igual forma si en `F` el par ordenado cuya primera componente es `X` tiene como segunda componente algo que no es `Y`, la unificación también fallará (lo que es equivalente a $f\ x \neq y$). La variable `G` es una variable existencial; se interpreta como "existe `G` tal que. . . ".

La definición de `applyTo` es la siguiente:

$$applyTo(f,x,y) \widehat{=} f = \{(x,y)/g\} \wedge (x,y) \notin g \wedge comp(\{(x,x)\},g,\emptyset)$$

La justificación es la siguiente. Si sabemos que $x \in \text{dom} f$ entonces existen Y y G tales que $f = \{(x,y)/g\} \wedge (x,y) \notin g$, por lo que analizamos más arriba. Si además estamos diciendo que podemos aplicar f a x es porque en f hay un único par ordenado cuya primera componente es x. Notar que no estamos diciendo que f es una función, solo estamos diciendo que f es función *localmente* en x (tal vez lo sea en otros puntos de su dominio pero por el momento no lo sabemos). Decir que en f hay un único par ordenado cuya primera componente es x es lo mismo que decir que en g no hay ningún par ordenado cuya primera componente sea x. Esto lo decimos usando el operador de composición de relaciones binarias `comp` (ver Tabla 2) mediante el predicado $comp(\{(x,x)\},g,\emptyset)$. En efecto, este predicado dice que cuando se compone $\{(x,x)\}$ con g el resultado es el conjunto vacío. Esto puede darse por dos motivos: g es la relación binaria vacía, en cuyo caso obviamente no hay ningún par ordenado con primera componente x; o g es no vacía pero no hay ningún par ordenado que componga con x, lo que es lo mismo que decir que x no pertenece al dominio de g. Lo mismo se podría haber codificado con $dom(g,D) \wedge x \notin D$ pero es más ineficiente porque requiere calcular el dominio de g.

De esta forma, al usar `applyTo` en lugar de unificación estamos pidiendo un poco más porque pedimos que haya un único par ordenado cuya primera componente sea x. Cuando usamos unificación solo pedimos que al menos haya un par ordenado de la forma (x,y).

Otra definición posible de `applyTo` es la siguiente:

$$applyTo(f,x,y) \widehat{=} comp(\{(x,x)\},f,\{(x,y)\})$$

Preferimos la primera porque, nuevamente, es más eficiente que esta última.

3.4. Traducción de expresiones aritméticas

En general las expresiones aritméticas de Z se traducen de forma directa a {*log*}, aunque hay algunas excepciones. Los símbolos $\leq$ y $\geq$ se traducen como `=<` y `>=`, respectivamente. El operador $\neq$ se traduce como `neq`. Los operadores aritméticos son los habituales: `+`, `-`, `*`, `div` y `mod`.

Una igualdad de la forma $x' = x+1$ se traduce como `X_ is X + 1` (es decir, en igualdades aritméticas, no hay que usar `=` sino `is`). Más aun, si en Z tenemos $A = \{x, y-4\}$ (A, x e y variables) tenemos que traducirlo como: `A = {X,Z} & Z is Y - 4`, donde `Z` debe ser una variable no usada en el predicado. El problema es que ni {*log*} ni Prolog evalúan expresiones aritméticas a menos que el programador lo pida usando el operador `is` o el operador `=:=`. Es decir si en {*log*} ejecutamos `5 + 1 in {6}`, la respuesta será `no` porque {*log*} intentará determinar si `5 + 1 = 6` sin evaluar `5 + 1` (es decir la considera, básicamente, como una cadena de caracteres). Por el contrario, si ejecutamos `Y is 5 + 1 & Y in {6}` {*log*} responderá `true`. Igualdades tales como `5 + 1 = 4 + 2` se pueden escribir como `X is 5 + 1 & Y is 4 + 2 & X = Y` pero es más conveniente así `5 + 1 =:= 4 + 2`. Tal como con `is`, {*log*} fuerza la evaluación de las dos expresiones involucradas en `=:=`.

Lo mismo aplica al predicado `neq`: para {*log*} `5 + 1 neq 6` es verdadero. En consecuencia debemos escribir: `H is 5 + 1 & H neq 6`. Sin embargo, con los operadores de orden no hay problema: `5 + 1 > 4` es verdadera.

Operador	Predicado {*log*}	Significado
conjunto	`set(A)`	A es un conjunto
igualdad	`A = B`	$A = B$
pertenencia	`x in A`	$x \in A$
unión	`un(A,B,C)`	$C = A \cup B$
intersección	`inters(A,B,C)`	$C = A \cap B$
diferencia	`diff(A,B,C)`	$C = A \setminus B$
subconjunto	`subset(A,B)`	$A \subseteq B$
subconjunto estricto	`ssubset(A,B)`	$A \subset B$
conjuntos disyuntos	`disj(A,B)`	$A \parallel B$
cardinalidad	`size(A,n)`	$\|A\| = n$
máximo de un conjunto	`smax(A,n)`	n es el máximo del conjunto A
mínimo de un conjunto	`smax(A,n)`	n es el mínimo del conjunto A
	Negaciones	
igualdad	`A neq B`	$A \neq B$
pertenencia	`x nin A`	$x \notin A$
unión	`nun(A,B,C)`	$C \neq A \cup B$
intersección	`ninters(A,B,C)`	$C \neq A \cap B$
diferencia	`ndiff(A,B,C)`	$C \neq A \setminus B$
subconjunto	`nsubset(A,B)`	$A \not\subseteq B$
conjuntos disyuntos	`ndisj(A,B)`	$A \not\parallel B$

Tabla 1: Operadores de la teoría de conjuntos disponibles en {*log*}

Para indicar que el conjunto `A` es un subconjunto de los naturales se debe hacer mediante una cuantificación universal restringida (ver Sección 3.7.1):

```
foreach(X in A, 0 =< X)
```

De todas formas, en muchas situaciones, esta restricción corresponde más un invariante. Si es un invariante se debería probar que efectivamente lo es y no establecer por definición que lo es (recordar la diferencia entre invariantes por definición y por demostración vista en Ingeniería de Software I [1]).

3.5. Traducción de los operadores de la teoría de conjuntos

Los operadores de la teoría de conjuntos (incluyendo relaciones binarias, funciones parciales y secuencias) se traducen según lo muestran las Tablas 1, 2 y 4.

El operador de cardinalidad solo acepta una variable o una constante como segundo argumento. Es decir que si ejecutamos `size(A,X + 1)` {*log*} responde `no`; en cambio si ejecutamos `size(A,Y) & Y is X + 1` (`Y` debe ser una variable no usada en el predicado) la respuesta es `true` porque la fórmula es satisfacible. {*log*} también responderá `no` si ejecutamos `size(A,Y) & Y = X + 1`.

Lo mismo ocurre con los operadores `smax` y `smin`, solo aceptan una variable o una constante como segundo argumento.

Operador	Predicado {*log*}	Significado
relación binaria	`rel(R)`	R es una relación binaria
función parcial	`pfun(R)`	R es una función parcial
aplicación de función	`applyTo(f,x,y)`	$f(x) = y$
dominio	`dom(R,A)`	$\mathrm{dom}\, R = A$
rango	`ran(R,A)`	$\mathrm{ran}\, R = A$
composición	`comp(R,S,T)`	$T = R \circ S$
inversa	`inv(R,S)`	$S = R^{-1}$
restricción de dominio	`dres(A,R,S)`	$S = A \lhd R$
anti-restricción de dominio	`dares(A,R,S)`	$S = A \lhd\!\!\!\cdot\, R$
restricción de rango	`rres(A,R,S)`	$S = R \rhd A$
anti-restricción de rango	`rares(A,R,S)`	$S = R \rhd\!\!\!\cdot\, A$
actualización	`oplus(R,S,T)`	$T = R \oplus S$
imagen relacional	`rimg(R,A,B)`	$B = R[A]$
Negaciones		
Todas las negaciones se escriben anteponiendo una letra `n` al operador correspondiente. Por ejemplo la negación de `dom(R,A)` es `ndom(R,A)`, la de `dares(A,R,S)` es `ndares(A,R,S)`, etc.		

Tabla 2: Operadores relacionales disponibles en {*log*}

3.6. Traducción de listas

La teoría de listas es, en general, indecidible. {*log*} intenta trabajar con fragmentos decidibles de la teoría de conjuntos—básicamente porque en ese caso se pueden hacer demostraciones automáticas. Por lo tanto, en general, no es posible traducir a {*log*} cualquier fórmula que involucre listas. De todas formas, hay algunos fragmentos de la teoría de listas que son decidibles. Vamos a ver cómo usar dos de ellos.

3.6.1. Conjuntos ordenados

Un *conjunto ordenado* ('ordered set', en inglés) es una función parcial con dominio en $\mathbb{N}_1$. Por ejemplo, $\{(1,a),(3,b)\}$ es un conjunto ordenado—pero no es una secuencia porque el dominio no es de la forma $1\,..\,n$ para algún n. Los conjuntos ordenados mantienen los elementos del rango ordenados y permiten repeticiones. En efecto, en $\{(1,a),(3,b)\}$ podemos decir que a es el primer elemento del conjunto porque 1 es menor a 3 y b el segundo porque 3 es mayor que 1. El conjunto ordenado $\{(1,b),(3,b)\}$ repite el elemento b. Por otro lado, el conjunto ordenado $\{(1,a),(3,b)\}$ representa la lista $\langle a,b\rangle$ al igual que el conjunto ordenado $\{(2,a),(4,b)\}$. Es decir que de cierta forma $\{(1,b),(3,b)\}$ y $\{(2,a),(4,b)\}$ son iguales, pero de cierta otra no lo son. Si pensamos que un conjunto ordenado representa una lista entonces $\{(1,b),(3,b)\}$ y $\{(2,a),(4,b)\}$ son iguales, pero como conjuntos claramente no lo son. Esto implica que la igualdad de conjuntos *no* sirve para determinar si dos conjuntos ordenados representan la misma lista o no. Podemos decir que la igualdad de conjuntos *no* determina si dos conjuntos ordenados son iguales o no.

La idea de los conjuntos ordenados es que se pueden definir algunas de las operaciones de listas (o secuencias) de manera tal que una clase muy expresiva de fórmulas pertenezca a un

fragmento decidible[4]. Más aun, es más bien fácil determinar la decidibilidad de ese fragmento habiendo determinado la decidibilidad de otros fragmentos de la teoría de conjuntos. Por el mismo motivo la expresividad de los conjuntos ordenados es mucho más acotada que la expresividad de la teoría de listas pero sigue siendo útil en muchos problemas donde se necesita algo más que conjuntos pero no tanto como listas. Si en esos problemas usamos conjuntos ordenados en lugar de listas tenemos la expresividad necesaria manteniendo la decidibilidad lo que implica la posibilidad de realizar demostraciones automáticas—que a su vez son útiles cuando queremos verificar formalmente especificaciones o programas.

Para usar conjuntos ordenados en {*log*} tienen que cargar la biblioteca `osets.slog`:

```
{log}=> add_lib('osets.slog').
```

Los operadores disponibles en `osets.slog` se listan en la Tabla 3. Todos los operadores *asumen* que los argumentos `S`, `T` y `U` verifican `oset/1` o `ioset/1`. Esto significa que en algunos casos van a tener que incluir `oset/1` o `ioset/1` para uno o más argumentos porque de lo contrario los operadores pueden no funcionar del modo esperado. Por ejemplo, `del({[a,b],[1,x]},x,T)` va a asociar `T` con `{[a,b]}` a pesar de que `{[a,b],[1,x]}` no es un conjunto ordenado porque el dominio no es un subconjunto de $\mathbb{N}_1$. Si el control de tipos está activado algunos de estos errores van a aparecer como errores de tipos. Sin embargo, en casos como `del({[1,b],[1,x]},x,T)` el control de tipos no es suficiente. En este caso {*log*} va a asociar `{[1,b]}` con `T` a pesar de que `{[1,b],[1,x]}` no es un conjunto ordenado porque no es una función.

Si están haciendo una demostración que involucra operadores de la Tabla 3 y la respuesta que obtienen es satisfactoria entonces no hace falta agregar predicados `oset/1` o `ioset/1`. En caso contrario, podría ser que la respuesta inesperada se deba a que falta alguna restricción `oset/1` o `ioset/1`. Agregar estas restricciones podría hacer que las demostraciones requieran más tiempo. Por ese motivo puede ser conveniente empezar sin esas restricciones y agregarlas cuando es imprescindible.

La restricción `ioset/1` establece que el conjunto ordenado es inyectivo; es decir, no hay o no admite repeticiones de elementos en el rango del conjunto ordenado. En este caso los operadores de la Tabla 3 son deterministas. Por ejemplo:

```
{log}=> next({[2,a],[5,b],[6,c],[9,a],[11,d]},a,B).   % no ioset

B = b  ;  B = d     % dos soluciones, no determinista

{log}=> next({[2,a],[5,b],[6,c],[9,k],[11,d]},a,B).   % ioset

B = b               % una solución, determinista
```

Lo mismo ocurre con `oin/3`.

La biblioteca provee algunos operadores que usan los índices de los elementos. En general, no hay un invariante significativo sobre los índices que se pueda garantizar en todas las operaciones sobre conjuntos ordenados. Por el contrario cuando trabajamos con listas se sabe que el dominio de una lista (o sea los índices) es siempre un intervalo de la forma $[1,n]$ donde n es la longitud

[4]De ahora en más es posible que digamos que los conjuntos ordenados son una teoría o fragmento decidible. En realidad el fragmento decidible es más pequeño pero lo expresamos de esa forma para simplificar la exposición.

Operador	Predicado {*log*}	Significado
	Tipos	
conjunto ordenado	`oset(S)`	S es un conjunto ordenado
conjunto ordenado inyectivo	`ioset(S)`	S es un conjunto ordenado inyectivo, no admite repeticiones
	Constructores	
añade un elemento	`add(S,Y,T)`	$T = S \cup \{max(\mathrm{dom}\, S) + 1 \mapsto Y\}$
elimina un elemento	`del(S,Y,T)`	$T = S \rhd \{Y\}$
elimina primera ocurrencia	`delf(S,Y,T)`	$T = S \setminus \{min(\mathrm{dom}\, S \rhd \{Y\}) \mapsto Y\}$
elimina última ocurrencia	`dell(S,Y,T)`	$T = S \setminus \{max(\mathrm{dom}\, S \rhd \{Y\}) \mapsto Y\}$
cola	`tail(S,T)`	$T = tail(S)$
frente	`front(S,T)`	$T = front(S)$
extraer	`extract(S,M,N,T)`	$T = M..N \lhd S$
filtro	`filter(S,SY,T)`	$T = S \rhd SY$
concatena	`concat(S,T,U)`	$U = S \frown T$
prefijo antes de Y	`prefix(S,Y,T)`	$T = \{(a,b) \in S \mid a < min(\mathrm{dom}\, S \rhd \{Y\})\}$
sufijo después de Y	`suffix(S,Y,T)`	$T = \{(a,b) \in S \mid max(\mathrm{dom}\, S \rhd \{Y\}) < a\}$
intercambia A con B	`switch(S,A,B,T)`	$T = S \oplus \{S^{-1}(A) \mapsto B, S^{-1}(B) \mapsto A\}$
	Observadores	
igualdad	`oseq(S,T)`	S y T representan la misma lista
desigualdad en `ioset`	`nioseq(S,T)`	S y T no representan la misma lista inyectiva
pertenencia	`oin(Y,S)`	Y es un elemento del rango de S
pertenencia con índice	`oin(M,Y,S)`	(M,Y) es un elemento de S
no pertenencia	`noin(Y,S)`	Y no es un elemento del rango de S
primer elemento	`head(S,Y)`	Y primer elemento de la 'lista' S
primer elemento con índice	`head(S,M,Y)`	(M,Y) primer elemento de S
último elemento	`last(S,Y)`	Y último elemento de la 'lista' S
último elemento con índice	`last(S,M,Y)`	(M,Y) último elemento de S
más adelante	`after(S,A,B)`	B está más adelante de A en la 'lista' S
siguiente	`next(S,A,B)`	B siguiente elemento de A en la 'lista' S
siguiente con índices	`next(S,M,A,N,B)`	(N,B) siguiente elemento de (M,A) en S
ordenado	`sorted(S)`	el rango de S está ordenado
duplicado	`duplicated(S,Y)`	Y está repetido en la 'lista' S

Tabla 3: Operadores disponibles para conjuntos ordenados. El significado se da en términos de los operadores de listas aunque los argumentos no son listas; en consecuencia el resultado tampoco es una lista. Para tener la semántica precisa sugerimos leer el código fuente de cada operador.

de la lista. Intentar establecer un invariante semejante para los conjuntos ordenados conlleva problemas de decidibilidad.

En particular, no es posible acceder a un elemento dada su posición porque en general no conocemos cuáles son las posiciones disponibles en un conjunto ordenado. Haciendo `add({},a,T)` sabemos que `a` tiene índice 1; si luego hacemos `add(T,b,U)` sabemos que `a` tiene índice 1 y `b` tiene índice 2; pero si hacemos `tail(U,V)` en `V` el índice 1 no está más[5]. Por lo tanto, no podemos garantizar el acceso dado un índice. Una posibilidad es calcular el índice de y invocando `oin(`m`,`y`,`S`)`. El problema es que si S no es inyectivo, el predicado es no determinista. Podemos evitar el no determinismo empezando por `head(`S`,`m`,`y`)` para obtener el primer 'punto' de S y luego invocar `next(`S`,`m`,`y`,`n`,`b`)` para encontrar (n, b), el segundo punto de S, y así recorrer determinística y ordenadamente S. También podemos recorrer S en orden inverso empezando por `last(`S`,`m`,`y`)`. En cualquier caso disponemos solo de un recorrido secuencial sobre un conjunto ordenado si queremos evitar el no determinismo.

Esta limitación de los conjuntos ordenados hace que sea bastante fácil demostrar que la teoría es decidible. Esta limitación es, tal vez, la diferencia más significativa con las listas. De la misma forma y por el mismo motivo, los conjuntos ordenados no disponen de operaciones que calculen los primeros o últimos n elementos. Se dispone de una operación (`extract/4`) que extrae todos los elementos de un conjunto ordenado que están entre dos índices (m y n) pero el resultado puede ser muy diferente al del operador similar para listas. En efecto, si $m < n$ son dos índices del conjunto ordenado S, el resultado puede ser, por ejemplo, el conjunto vació porque S puede tener 'huecos'—cosa que es imposible que ocurra con las listas.

Recuerden que para determinar la igualdad entre dos conjuntos ordenados *no* pueden usar la igualdad de conjuntos (o sea el operador =). En cambio, tienen que usar `oseq/2` que es computacionalmente muy pesado. Esto significa que una demostración que lo involucre puede demorar mucho tiempo. Otro problema con la igualdad entre conjuntos ordenados es que no es posible escribir un predicado {*log*} que determine si dos conjuntos ordenados son diferentes. En otras palabras no es posible definir la negación de `oseq/2`. Sin embargo, es posible definir la negación cuando los conjuntos ordenados son inyectivos (`nioseq/2`). Esto es esperable porque los conjuntos ordenados inyectivos son menos expresivos que los no inyectivos.

Cuándo usar conjuntos ordenados. Si necesitamos orden o repeticiones no podemos usar conjuntos. En ese caso la primera elección deben ser los conjuntos ordenados. Si no es posible usarlos, entonces hay que recurrir a los arreglos (Sección 3.6.2). Los conjuntos ordenados son suficientemente expresivos si no tenemos que usar mucho los índices o posiciones de los elementos. Por ejemplo, si tenemos que modelar FIFOs o LIFOs, podemos usar conjuntos ordenados; no necesitamos listas. Si el problema pide saber si un elemento está o no antes que otro, podemos usar un conjunto ordenado. Esto sirve cuando el requisito es mantener un orden cronológico pero no nos piden saber qué elemento se guardó en tal momento, aunque nos pueden pedir saber si un elemento se guardó antes que otro o no. Si podemos evitar las repeticiones manteniendo la inyectividad de los conjuntos ordenados la lógica se vuelve más simple.

[5] Si `tail` fuese el operador de listas `V` sería una lista por lo que 1 sería índice en `V`.

Si tienen que usar fórmulas más o menos complejas que involucren conjuntos ordenados, consulten con el docente.

3.6.2. Arreglos

Para poder trabajar con arreglos es necesario cargar el archivo `array.slog`:

```
{log}=> add_lib('osets.slog').
```

La Tabla 4 lista los operadores disponibles en {*log*} para trabajar con arreglos. Si bien la teoría de arreglos es más expresiva que la de conjuntos ordenados, para mantener la decidibilidad se debe reducir su expresividad. De aquí que halla menos operadores en la Tabla 4 que en la 3. Por otro lado, la decidibilidad de la teoría de arreglos en {*log*} está aun en estudio. Para usar los operadores de la Tabla 4 tienen que tener en cuenta las restricciones que se listan a continuación. El usuario es responsable de que estas restricciones se verifiquen en las fórmulas que escribe; {*log*} no las controla. En caso de que alguna de estas restricciones no se verifique el comportamiento de {*log*} puede dar lugar a respuestas erróneas.

- Si A está declarado como un arreglo entonces no se puede calcular la cardinalidad de A ni de nada que se relacione con A. Por ejemplo, la siguiente fórmula puede generar comportamientos incorrectos en {*log*}.

  ```
  arr(A,N) & ran(A,R) & size(R,K) & K < N.
  ```

  La restricción se viola porque `A` es un arreglo y estamos calculado la cardinalidad de su rango.

  Tener en cuenta que la cardinalidad de A está siempre disponible a través del predicado `arr/2`.
- En todos los operadores, `n` y `k` solo pueden ser números o variables (no pueden ser expresiones).
- Siempre que los parámetros sean `A` y `n`, significa que `A` debe ser un arreglo de longitud `n`. O sea que tiene que haber un predicado `arr(A,n)` en la fórmula.
- En `add(A,n,e,B)`, se supone `arr(B,n+1)`.
- En `prefix(A,n,k,B)`, se supone `arr(B,k)`.
- En los operadores denominados "pseudo", `B` es una función pero no necesariamente es un arreglo. En general `B` no será un arreglo porque el dominio no es un intervalo de la forma $[1,m]$ para algún m.
- En `filter(A,S,B)`, `S` solo puede ser `{}`, una variable, un conjunto extensional o un intervalo (no puede ser un RIS).
- En `extract(A,S,B)`, `S` solo puede ser `{}`, una variable o un conjunto extensional (no puede ser un RIS ni un intervalo).

Si tienen que usar fórmulas más o menos complejas que involucren arreglos o si tienen que usar otros operadores que no están en la Tabla 4, consulten con el docente.

Operador	{*log*}	Significado (en Z)
arreglo	`arr(A,n)`	`A` es un arreglo de longitud `n`
arreglo extensional	`[1,a],[2,b],...,[n,z]`	$\langle a, b, \ldots, z \rangle$
cabecera	`head(A,e)`	$e = head\ A$
último	`last(A,n,e)`	$e = last\ A$
agregar	`add(A,n,e,B)`	$A = B \frown \langle e \rangle$
prefijo	`prefix(A,n,k,B)`	$B = 1..k \upharpoonright A$
sumatoria	`arrsum(A,n,k,s)`	$s = \sum_{i=1}^{k} A(i)$
ordenado	`sorted(A,n)`	A está ordenado
pseudo-sufijo	`suffix(A,n,k,B)`	$B = k..n \lhd A$
pseudo-filtrado	`filter(S,A,B)`	$B = S \lhd A$
pseudo-extracción	`extract(A,S,B)`	$B = A \rhd S$

Tabla 4: Operadores de arreglos disponibles en {*log*}

3.7. Traducción de operadores lógicos

En {*log*} están disponibles la conjunción (`&`), disyunción (`or`), implicación (`implies`), un operador tipo *let* (`let`) y negación (`neg`), entre otros conectores lógicos (ver Secciones 3.4 y 3.5 del manual del usuario de {*log*} para la lista completa y otras consideraciones importantes).

La negación (`neg`) hay que usarla con cuidado porque, como se explica en la sección 3.5 del manual, {*log*} no siempre es capaz de calcular bien la negación de una fórmula. En general, `neg` funciona como se espera cuando la fórmula a negar no contiene variables cuantificadas existencialmente *en su interior*. Por ejemplo, el siguiente predicado establece que `Min` es el mínimo en `S`[6]:

```
setmin(S,Min) :- Min in S & subset(S,int(Min,Max)).
```

`neg` no calculará correctamente la negación de `setmin(S,Min)` porque `Max` es una variable existencial dentro de la fórmula (porque es una variable que está en la fórmula pero no es un argumento de la cabecera de la cláusula). Más precisamente, si definimos la cláusula `n_setmin` como sigue:

```
n_setmin(S,Min) :- neg(Min in S & subset(S,int(Min,Max))).
```

esta no corresponde a ¬ **`setmin(S,Min)`** porque `neg` no calculará la negación (correcta) de su argumento ya que este contiene a `Max`. `neg` va a calcular una fórmula pero no será la que estamos esperando.

Esto significa que en algunos casos vamos a tener que calcular la negación de una fórmula a mano. Para estas situaciones {*log*} provee la negación de cada uno de sus predicados predefinidos. En efecto, por caso, si queremos traducir $\neg\ x \in A$ lo podemos hacer como[7] `neg(X in A)` o `X nin A`. De igual forma, $\neg\ A = b$ se traduce como `neg(A = b)` o `A neq b`. En general, para cada operador de conjuntos y relacional existe un predicado que implementa su negación. Por ejemplo, la fórmula Z $A \not\subseteq B$ se traduce como `neg(subset(A,B))` o `nsubset(A,B)`. Las Tablas 1 y 2 incluyen las negaciones de todos los predicados de la teoría de conjuntos.

[6] Esta *no* es la implementación del operador `smin` mencionado en la Tabla 1.

[7] Siempre teniendo en cuenta la forma de traducir variables, constantes y expresiones.

Usamos `neg` para traducir este esquema Z:

WithdrawE

$\Xi Bank$
$n? : NIC$
$a? : MONEY$
$msg! : MSG$

$\neg\ (n? \in \mathrm{dom}\, sa \land a? \leq sa(n?) \land a? > 0)$
$msg! = error$

de la siguiente forma (asumimos que el esquema *Bank* declara solo la variable *sa*):

```
withdrawE(Sa,N,A,Msg_o) :-
  dom(Sa,D) &
  applyTo(Sa,N,Y) &
  neg(N in D & A =< Y & A > 0) &
  Msg_o = error.
```

Noten que `dom(Sa,D)` y `applyTo(Sa,N,Y)` van afuera de `neg` porque, de otra forma, `D` e `Y` habrían sido variables cuantificadas existencialmente dentro de la fórmula y, en consecuencia, `neg` no hubiera calculado la negación correcta. Tengan en cuenta que esas dos variables no están presentes en *WithdrawE*; las tuvimos que introducir en {*log*} para darles un nombre a las expresiones $\mathrm{dom}\, sa$ y $sa(n?)$. En otras palabras, hubiera estado mal poner `dom(Sa,D)` y `applyTo(Sa,N,Y)` dentro de `neg` porque esos predicados no deberían negarse. Por ejemplo, `dom(Sa,D)` dice que `D` es el (nombre del) dominio de `Sa`: no tiene sentido negar esto porque lo que estamos haciendo es *definir* que `D` es eso. Esta situación aparece frecuentemente cuando una especificación Z se traduce a {*log*} debido al hecho de que Z usa expresiones para lo que en {*log*} se expresa con predicados.

{*log*} también ofrece el predicado `let` con el cual podemos escribir `withdrawE` así:

```
withdrawE(Sa,N,A,Msg_o,Sa_) :-
  neg(
    let([D,Y],
        dom(Sa,D) & applyTo(Sa,N,Y), N in D & A =< Y & A > 0
    )
  ) &
  Msg_o = error &
  Sa_ = Sa.
```

En este caso {*log*} reescribe `neg` de la siguiente forma:

```
dom(Sa,D) & applyTo(Sa,N,Y) & neg(N in D & A =< Y & A > 0)
```

que es exactamente la fórmula que escribimos incialmente.

Recomendamos leer la sección 3.5 del manual del usuario de {*log*} para aprender más sobre el predicado `let`.

3.7.1. Cuantificadores

En general el cuantificador existencial no es necesario escribirlo de forma explícita porque la semántica de los programas {*log*} es una cuantificación existencial. Por ejemplo si en Z tenemos:

$$\exists x : \mathbb{N} \mid x \in A$$

lo traducimos simplemente como:

```
0 =< X & X in A
```

porque la semántica del predicado {*log*} es, esencialmente, una cuantificación existencial. De todas formas, si eventualmente va a ser necesario calcular la negación del predicado donde escribimos `0 =< X & X in A`, entonces es necesario escribir el cuantificador existencial:

```
exists(X in A, 0 =< X)
```

La cosa cambia cuando se trata de cuantificadores universales. En {*log*} solo existen los cuantificadores universales restringidos (RUQ). Un RUQ es una fórmula de la forma:

$$\forall x \in A : P(x)$$

donde A es un conjunto y P es una proposición que depende de x. En {*log*} los RUQ se codifican de la siguiente forma: `foreach(x in A, P(x))`, aunque hay formas más complejas y expresivas

Recuerden que un uso apropiado del lenguaje Z reduce al mínimo la necesidad de fórmulas cuantificadas.

Los RUQ y REQ en {*log*} son más o menos complejos así que si necesitan usarlos primero lean el capítulo 6 del manual de {*log*} y en última instancia pidan ayuda al docente.

3.8. Traducción de definiciones axiomáticas

No existe una forma estándar y única de traducir las definiciones axiomáticas de Z pues estas son muy generales y sirven para múltiples propósitos. Por este motivo mostraremos cómo traducir las formas más habituales de definiciones axiomáticas.

Uno de los problemas al traducir definiciones axiomáticas es que son objetos globales de una especificación Z y {*log*} no provee un mecanismo simple para definir y usar objetos globales.

En general, las variables declaradas en una definición axiomática se declaran por medio de la palabra reservada `parameters` la cual debe ubicarse antes de la declaración de las variables de estado (que se hace con `variables`). Luego, los predicados de una definición axiomática se codifican en una cláusula que debe precederse con la declaración `axiom`. Todos los predicados declarados como axiomas se deben escribir luego de `variables` y antes del primer invariante. Los axiomas pueden depender únicamente de las variables declaradas en `parameters`.

De todas formas, si la intención de la definición axiomática es más bien introducir algo similar a una constante entonces se puede usar una constante Prolog o {*log*}.

Veamos algunos casos típicos.

Especificación Z

$$root : USR$$

Código {*log*}

Como la intención es declarar una constante que representa al usuario *root*, podemos usar la palabra `root` en las operaciones. Si tenemos el tipo `usr` y queremos que `root` sea un elemento especial, entonces una alternativa es definir un tipo suma en lugar de un tipo básico: `sum([root,u(usr)])`.

Especificación Z

$$adm : \mathbb{P}\, USR$$
$$adm = \{root, sec\}$$

Código {*log*}

En este caso podemos usar las constantes `root` y `sec` como en el caso anterior y si necesitamos el conjunto *adm* podemos usarlo de forma explícita `{root,sec}` o podemos definir un predicado.

```
dec_p_type(adm(set(usr))).
adm(X) :- X = {root,sec}.
```

Cuando queremos usar el conjunto lo podemos hacer de esta forma:

```
crearUsr(...,A,U,...) :- ... & adm(Adm) & A in Adm...
```

O sea que así chequeamos si `A` es uno de los administradores.
Lo que ocurre en este caso es que la variable `Adm` *unifica* con la variable `X` (que es el parámetro formal de la definición de `adm`) lo que implica que vale que `Adm` es igual a `{root,sec}`.

Especificación Z

$$adm : \mathbb{P}\, USR$$
$$adm \neq \emptyset$$

Código {*log*}

En este caso *adm* es un cierto conjunto no vacío de usuarios; la intención no es declarar una constante sino un parámetro de la especificación Z. En este sentido la definición axiomática actúa como una variable global. En este caso usamos `parameters` y `axiom`.

```
parameters([Adm]).

axiom(adm).
dec_p_type(adm(set(usr))).
adm(Adm) :- Adm neq {}.
```

Especificación Z

$$checksum : \operatorname{seq} BYTE \rightarrow BYTE$$

Código {*log*}

En {*log*} no existen las secuencias como en Z pero se las puede aproximar con un conjunto. Así que primero declaramos un parámetro.

```
parameters([Checksum]).
```

Luego usamos un axioma para decir que `Checksum` es una función.

```
axiom(checksum_pf).
dec_p_type(checksum_pf(set(byte),byte)).
checksum_pf(Checksum) :- pfun(Checksum).
```

Especificación Z

$$maximo : \mathbb{Z}$$

$$0 \leq maximo \leq 100$$

Código {*log*}

En este caso *maximo* es un parámetro así que los declaramos como tal.

```
parameters([Maximo]).
```

Luego usamos un axioma para indicar el rango de valores que puede tomar el parámetro.

```
axiom(rango_maximo).
dec_p_type(rango_maximo(int)).
rango_maximo(Maximo) :- 0 =< Maximo & Maximo =< 100.
```

4. Simulación de prototipos {*log*}

Cuando una especificación Z se traduce a un programa {*log*}, lo que tenemos, desde cierto punto de vista, es un prototipo del sistema que queremos implementar. Efectivamente, un

programa {*log*} es eso, un programa, pero es un programa que se obtiene de manera casi directa a partir de una especificación Z[8]. Además es un programa que en general no satisfará los requisitos de desempeño de un programa de producción por lo que es en realidad un prototipo del programa final. El punto es que {*log*} permite obtener un prototipo casi sin tener que trabajar extra, solo es necesario escribir la especificación y luego traducirla. Esto permitiría validar tempranamente los requerimientos con el usuario final.

Validar los requerimientos significa poder ejecutar el prototipo junto al usuario para que este pueda observar las salidas y efectos producidos por el prototipo, y así determinar si cumple con sus expectativas o no. Si bien los prototipos con los que trabajaremos en esta actividad no son los ideales, sirven como para dar una idea de la potencialidad de la combinación entre Z y {*log*}.

Dado que vemos a los programas {*log*} como prototipos hablamos más de *simulaciones* o *animaciones* que de *ejecuciones*, aunque en términos técnicos no es más que ejecutar un programa. El término *simulación* se suele usar en el contexto de *modelos* (e.g. modelado y simulación, o *modeling and simulation*). Como nuestros programas {*log*} son traducciones casi mecánicas de especificaciones Z entonces podemos decir que son *modelos* de los requerimientos del usuario. Por otro lado, el término *animación* se suele usar en el contexto de especificaciones formales. En ese sentido, un programa {*log*} puede verse como una especificación ejecutable. De hecho, como veremos un poco más adelante, un programa {*log*} tiene propiedades que lo acercan a las especificaciones y a los modelos y que en general no las tienen los programas escritos en lenguajes de programación imperativos (y hasta funcionales).

Sea ejecución, simulación o animación, la idea básica es dar entradas al programa, modelo o especificación y observar las salidas o efectos producidos. De todas formas mostraremos que {*log*} ofrece ciertas posibilidades más allá de esta idea básica.

4.1. El entorno de simulación Next

{*log*} ofrece Next, un entorno de simulación que simplifica la ejecución de escenarios funcionales sobre máquinas de estado. En Next se pueden correr simulaciones *deterministas* y *totalmente especificadas*. Una simulación es o está totalmente especificada cuando al finalizar solo hay variables a la izquierda de las igualdades que devuelve {*log*} y no hay una lista de restricciones en la respuesta. Para ejecutar simulaciones más complejas, llamadas *ejecuciones simbólicas*, ver la Sección 4.2.

Ejecutar una simulación en Next sobre una máquina de estados consiste de:

- Establecer el estado de partida para la simulación.
- Definir la secuencia de operaciones (transiciones de estado) que se deben ejecutar dando valores a las variables de entrada de cada operación.

Veamos un ejemplo de simulación en Next. Consideramos que el prototipo de la agenda de cumpleaños está almacenado en el archivo `bb.slog`. Empezamos ejecutando el intérprete Prolog desde una terminal de comandos y desde el directorio donde instalamos {*log*}[9].

[8] De hecho la traducción podría automatizarse, aunque probablemente no de forma completa y seguramente una traducción manual producirá un prototipo algo más eficiente.

[9] El nombre del ejecutable del intérprete Prolog puede variar dependiendo del sistema operativo. Además de `swipl` está disponible `swipl-win` que ofrece algunas ventajas para el manejo de la terminal.

```
1 ~/setlog$ swipl
2 ?- consult('setlog.pl').
3 ?- setlog.
4 {log}=> consult('bb.slog').
5 {log}=> vcg('bb.slog').
6 {log}=> consult('bb-vc.slog').
7 {log}=> initial >> addBirthday(Name:messi,Date:2406).
8
9 Final result is:
10 Known = {messi},
11 Birthday = {[messi,2406]}
```

El significado de cada línea es el siguiente:

1. Ejecutamos el intérprete de Prolog.
2. Cargamos el intérprete {*log*}.
3. Entramos en el intérprete {*log*}.
4. Cargamos el prototipo de la agenda de cumpleaños.
5. Ejecutamos el generador de condiciones de verificación (VCG) sobre la especificación de la agenda de cumpleaños. Este paso es necesario para poder usar NEXT. En la Sección 5 vamos a ver el VCG en detalle.
6. Cargamos el archivo que genera el VCG. NEXT necesita que ese archivo esté cargado.
7. Ejecutamos la simulación:

   ```
   initial >> addBirthday(Name:messi,Date:2406).
   ```

   donde:

   - `initial` es una palabra reservada que indica que la simulación se inicia en el estado inicial definido en la especificación. Recuerden que usamos la declaración `initial(birthdayBookInit)` para indicar que el predicado `birthdayBookInit` define el estado inicial de la especificación. En este sentido `initial` hace que NEXT invoque el predicado `birthdayBookInit`.
   - El operador >> se denomina *then* o *luego* o *después*.
   - Le decimos a NEXT que invoque el predicado `addBirthday` asumiendo que los argumentos `Name` y `Date` son de entrada y por lo tanto les asignamos los valores `messi` y `2406`, respectivamente. Los términos como `Name:messi` se llaman *asignaciones*. A la izquierda de una asignación va el nombre de un parámetro de entrada de la operación.
     El estado de partida para ejecutar `addBirthday` es el estado que definió `initial`. En este caso la secuencia de operaciones que se debe ejecutar está formada por una sola invocación.

   Notar que el comando de la simulación termina con un punto.
8. En las líneas 9 a 11 NEXT muestra el resultado de la simulación. El resultado de la simulación es el último valor de las variables de estado y de salida.

Si queremos agregar dos o más personas a la agenda podemos hacerlo de la siguiente forma:

```
{log}=> initial >>
        addBirthday(Name:[[messi,pele]],Date:[[2406,2310]]).

Final result is:
Known = {messi,pele},
Birthday = {[messi,2406],[pele,2310]}
```

Las listas como `[[messi,pele]]` se denominan *listas dobles*. En una simulación podemos combinar listas dobles con asignaciones comunes.

```
{log}=> initial >> addBirthday(Name:[[messi,riquelme]],Date:2406).

Final result is:
Known = {messi,riquelme},
Birthday = {[messi,2406],[riquelme,2406]}
```

Podemos pedirle a NEXT que nos muestre la traza completa de la simulación encerrando `initial` entre corchetes.

```
{log}=> [initial] >>
        addBirthday(Name:[[messi,pele]],Date:[[2406,2310]]).

Execution trace is:
Known = {},
Birthday = {}
  ----> addBirthday(Name:messi,Date:2406)
Known = {messi},
Birthday = {[messi,2406]}
  ----> addBirthday(Name:pele,Date:2310)
Known = {messi,pele},
Birthday = {[messi,2406],[pele,2310]}
```

De esta forma podemos ver que partimos del estado inicial, luego agregamos la fecha de cumpleaños de Messi y luego la de Pelé. Notar que el estado final de la traza coincide con el estado final mostrado cuando ejecutamos la misma simulación sin pedir la traza completa.

También podemos tomar valores de salida de un paso de la simulación y usarlos como valores de entrada para un paso ulterior.

```
{log}=> initial >>
        addBirthday(Name:[[messi,pele,riquelme]],
                    Date:[[2406,2310,2406]]
        ) >>
        findBirthday(Name:messi,Date) >> remind(Today:Date,Cards).

Final result is:
Known = {messi,pele,riquelme},
Birthday = {[messi,2406],[pele,2310],[riquelme,2406]},
Date = 2406,
Cards = {messi,riquelme}
```

Vean que la asignación `Today:Date` en `remid` usa `Date` que es un parámetro de salida de `findBirthday`. Esto hace que Next asigne a `Today` el valor generado por `findBirthday` para `Date`. Como `Date` vale `2406` entonces `Cards` vale `{messi,riquelme}` puesto que `remind` calcula la lista de personas que cumplen años en esa fecha.

Next también ofrece la posibilidad de chequear si uno o más invariantes se verifican luego de cada paso de una ejecución. Por ejemplo, en la siguiente simulación Next va a corroborar si `birthdayBookInv` se satisface luego de cada paso de la ejecución.

```
{log}=> [initial]:[birthdayBookInv] >>
        addBirthday(Name:[[messi,pele,riquelme]],
                    Date:[[2406,2310,2406]]
        ) >>
        findBirthday(Name:messi,Date) >>
        remind(Today:Date,Cards).
```

En este caso Next evalúa `birthdayBookInv` en el estado final generado luego de cada paso de la ejecución e informa al usuario si el invariante *no* se satisface—si el invariante se satisface no se informa nada. Si el invariante se satisface no significa que la operación lo preserve. O sea esto no es equivalente a demostrar un lema de invarianza, solo brinda un indicio positivo sobre la especificación. Por el contrario si el invariante *no* se satisface luego de algún paso de la ejecución sabemos sin dudas que la especificación tiene algún error.

Si queremos chequear el cumplimiento de más invariantes solo los tenemos que agregar a la lista.

> Para terminar de aprender a usar Next tienen que leer la sección 13.2.2 del manual del usuario de {*log*}.

4.2. Simulaciones simbólicas

Simular simbólicamente un programa significa ejecutarlo proveyéndole variables como entradas en lugar de constantes. Esto implica que el motor de ejecución debe ser capaz de operar simbólicamente con las variables para ir determinando sus valores a medida que el programa avanza. Al operar con variables, una ejecución simbólica puede mostrar propiedades más generales de un programa que cuando se lo ejecuta partiendo de constantes.

{*log*} es capaz de simular simbólicamente los prototipos, dentro de ciertos límites. En particular no se puede usar Next para hacer ejecuciones simbólicas. Las condiciones para que {*log*} pueda realizar simulaciones simbólicas son las siguientes[10]:

1. Si el programa {*log*} usa cardinalidad (`size`) no debe usar los operadores de la Tabla 2.

[10] Esta es una descripción informal y no del todo precisa de las condiciones para realizar simulaciones simbólicas. Las condiciones precisas son más complejas y muy técnicas. Los programas {*log*} con los que no se pueden hacer simulaciones simbólicas y que no cumplen las condiciones indicadas son infrecuentes en la clase de problemas con los que nos enfrentamos en este curso.

2. Todas las fórmulas aritméticas son lineales[11].

El código de la agenda de cumpleaños está dentro de los límites para que {*log*} pueda operar simbólicamente. Por ejemplo, empezando desde el estado inicial podemos invocar a `addBirthday` usando únicamente variables:

```
{log}=> birthdayBookInit(K,B) & addBirthday(K,B,N,C,K_,B_).

K = {},
B = {},
K_ = {N},
B_ = {[N,C]}
```

Observar lo siguiente:

1. Tenemos que incluir de forma explícita las variables de estado y usarlas para encadenar el estado de llegada de una operación con el de partida de la siguiente.
2. {*log*} pregunta si queremos más soluciones.
3. La respuesta contiene variables a la derecha de las igualdades (`N` y `C`) y puede contener la sección de restricciones.

No es el caso de la simulación anterior pero en general las simulaciones simbólicas tienden a dar respuestas más o menos complejas que a primera vista pueden ser difíciles de analizar. Una forma de simplificar el análisis es pedirle a {*log*} que genere soluciones concretas[12]. En ese caso podemos usar el comando `groundsol`.

```
{log}=> groundsol.
{log}=> birthdayBookInit(K,B) & addBirthday(K,B,N,C,K_,B_).

K = {},
B = {},
N = n1,
C = n0,
K_ = {n1},
B_ = {[n1,n0]}
```

Notar que ahora tenemos dos constantes, `n0` y `n1`. Cuando el comando `groundsol` está activo {*log*} sustituye las variables a la derecha de las igualdades por constantes de la forma `n`⟨*número*⟩ donde *número* empieza en cero. En otras palabras, cuando `groundsol` está activo las únicas variables en las respuestas que da {*log*} son las que están a la izquierda de las igualdades. `groundsol` se desactiva con el comando `nogroundsol`. Para saber más sobre `groundsol` recomendamos leer las secciones 3.2 y 12.7 del manual de {*log*}.

Ahora desactivamos `groundsol` y encadenamos otra invocación más a `addBirthday` usando otro juego de variables para las entradas y para encadenar estados-siguientes con estados-anteriores (e.g. `K1` y `B1`):

[11] Más precisamente, las expresiones enteras son sumas o restas de términos de la forma `x*y` con `x` o `y` constante.
[12] 'ground solutions', en inglés.

```
{log}=> birthdayBookInit(K,B) &
        addBirthday(K,B,N,C,K1,B1) & addBirthday(K1,B1,M,D,K_,B_).

K = {},
B = {},
K1 = {N},
B1 = {[N,C]},
K_ = {N,M},
B_ = {[N,C],[M,D]}
Constraint: N neq M
```

En este caso vemos que {*log*} devuelve como parte de la respuesta la sección de restricciones (i.e. `Constraint`). Si no recuerdan el significado de esta parte de la respuesta que vimos en el tutorial, lean la Sección 3.2 del manual del usuario.

En este ejemplo, claramente, la segunda invocación a `addBirthday` puede añadir la entrada `[M,D]` a la agenda si y solo si `M nin {N}` (i.e. $M \notin \{N\}$), lo cual vale si y solo si `M` es distinto de `N`. Pueden probar a ver cómo cambia la respuesta si esta simulación se ejecuta luego de activar `groundsol`.

{*log*} retorna una segunda solución de esta simulación simbólica:

```
K = {},
B = {},
K1 = {N},
B1 = {[N,C]},
M = N,
K_ = {N},
B_ = {[N,C]}
```

producto de considerar que `N` y `M` son iguales en cuyo caso la segunda invocación a `addBirthday` va por la rama de `nameAlreadyExists` por lo que `K_` y `B_` resultan iguales a `K1` y `B1`, lo cual también es un resultado posible.

Claramente las simulaciones simbólicas nos permiten sacar conclusiones más generales sobre el comportamiento del prototipo. El siguiente ejemplo nos permite ver en forma más amplia lo que acabamos de mencionar:

```
{log}=> birthdayBookInit(K,B) &
        addBirthday(K,B,N,C,K1,B1) & addBirthday(K1,B1,M,D,K2,B2) &
        findBirthday(K2,B2,W,X,K2,B2).
```

puesto que {*log*} considerará varios casos posibles en relación a si `M`, `N` y `W` son iguales o no. Por ejemplo, las siguientes son las tres primeras soluciones:

```
K = {},
B = {},
K1 = {N},
B1 = {[N,C]},
K2 = {N,M},
B2 = {[N,C],[M,D]},
```

```
W = N,
X = C
Constraint: N neq M

Another solution?  (y/n)
K = {},
B = {},
K1 = {N},
B1 = {[N,C]},
K2 = {N,M},
B2 = {[N,C],[M,D]},
W = M,
X = D
Constraint: N neq M

Another solution?  (y/n)
K = {},
B = {},
K1 = {N},
B1 = {[N,C]},
K2 = {N,M},
B2 = {[N,C],[M,D]}
Constraint: N neq M, N neq W, M neq W
```

En el primer caso considera `W = N` y por lo tanto `X` debe ser igual a `C`; el segundo es similar al primero; y en el tercero `W` no es ni `M` ni `N` y por lo tanto el valor de `X` puede ser cualquiera (¿es esto un error?). {*log*} retorna más soluciones, pero algunas de ellas se repiten[13].

Obviamente en las simulaciones simbólicas se pueden combinar variables con constantes. En general cuantas menos variables se usen menos soluciones habrá y más determinista será el comportamiento de {*log*}, pero se podrán sacar conclusiones menos generales.

4.2.1. Simulaciones simbólicas tipadas

Al ejecutar las simulaciones simbólicas anteriores no usamos el sistema de tipos ni la información de tipos que declaramos en la agenda de cumpleaños. En otras palabras {*log*} ignoró las sentencias `dec_p_type` que incluimos en `bb.slog`. Es decir que si había algún error de tipos en las simulaciones no nos enteramos porque {*log*} no ejecutó el *typechecker*. En este sentido {*log*} ejecutó todas las simulaciones en modo no tipado. En esta sección veremos brevemente cómo activar el chequeo de tipos y cómo afecta esto a las simulaciones. Recuerden leer la sección 12 del manual de {*log*} para más detalles sobre el sistema de tipos.

El chequeo de tipos se debe activar antes de consultar el programa que queremos usar mediante el comando `type_check`.

```
~/setlog$ swipl
```

[13] La repetición de soluciones es un efecto secundario de {*log*} que por cuestiones propias de la teoría de conjuntos es imposible de evitar en todos los casos.

```
?- consult('setlog.pl').
?- setlog.
log=> type_check.              % activamos el type checker
log=> consult('bb.slog').
```

De esta forma, cuando {*log*} ejecuta el comando `consult` invoca el chequeo de tipos y si hay errores de tipos se verá el mensaje correspondiente.

El chequeo de tipos se puede desactivar en cualquier momento mediante el comando `notype_check`.

Cuando el control de tipos está activo *todas* las variables deben estar tipadas.

```
{log}=> birthdayBookInit(K,B) & addBirthday(K,B,maxi,160367,K_,B_).

***ERROR***: type error: variable K has no type declaration
```

Entonces debemos declarar el tipo de todas las variables:

```
{log}=> birthdayBookInit(K,B) &
        addBirthday(K,B,name:messi,date:2406,K_,B_) &
        dec([K,K_],kn) & dec([B,B_],bb).

K = {},
B = {},
K_ = {name:messi},
B_ = {[name:messi,date:2406]}
```

En {*log*} las constantes del tipo básico `t` son de la forma `t:`⟨*término*⟩ donde *término* debe ser un átomo de Prolog o un número entero[14]. No confundir la constante `name:messi` de tipo `name` con la asignación en entorno NEXT `Name:messi`.

Si uno quiere chequear los tipos de un programa, por ejemplo de `bb.slog`, pero no quiere tener problemas con los tipos cuando hace simulaciones lo que puede hacer es desactivar el control de tipos luego de haber cargado el programa. De esta forma {*log*} controlará los tipos del programa pero aceptará fórmulas (o programas o simulaciones) no tipadas.

Claramente, en general, trabajar con simulaciones no tipadas es menos engorroso pero más peligroso porque podríamos invocar al programa con entradas que no respetan los tipos del programa lo que podría ocasionar fallas espurias. De todas formas una ventaja de las simulaciones tipadas es que en modo `groundsol` las soluciones usan constantes del tipo correcto.

```
{log}=> type_check.
{log}=> groundsol.
{log}=> dec([K,K_],kn) & dec([B,B_],bb) &
        dec(N,name) & dec(C,date) &
        birthdayBookInit(K,B) & addBirthday(K,B,N,C,K_,B_).

K = {},
```

[14] Los átomos de Prolog son cadenas de caracteres que comienzan con una minúscula y no tienen espacios (SWI-Prolog: atom/1).

```
K_ = {name:n1},
B = {},
B_ = {[name:n1,date:n0]},
N = name:n1,
C = date:n0
```

4.2.2. Simulaciones simbólicas que involucran aritmética entera

Como ya hemos mencionado, {*log*} es fundamentalmente una herramienta para trabajar con la teoría de conjuntos. Sin embargo, también es capaz de resolver fórmulas que contienen predicados sobre la aritmética entera. Para trabajar con los números enteros {*log*} usa dos herramientas externas conocidas como CLP(FD)[15] y CLP(Q)[16]. Cada una de estas herramientas tiene sus ventajas y desventajas (ver la sección 7 del manual de usuario de {*log*}).

Por defecto {*log*} usa CLP(Q). El usuario puede cambiar a CLP(FD) mediante el comando `int_solver(clpfd)` y volver a CLP(Q) mediante `int_solver(clpq)`.

CLP(FD) es capaz de hacer enumeraciones sobre los números enteros (técnicamente conocido como *enumeración* o *labeling*[17]) lo que permite generar interactivamente todas las soluciones posibles donde participan números enteros. Para que la enumeración funcione algunas de las variables enteras tienen que tener asociado un dominio finito. La variable entera N tiene asociado el dominio finito `int(`a`,`b`)` (a y b números enteros) si en la fórmula está el predicado N `in int(`a`,`b`)`. Para más detalles consulten el capítulo 7 del manual de {*log*}.

En general una simulación puede contener restricciones aritméticas enteras. Por ejemplo, si CLP(Q) está activo, la respuesta de {*log*} al ejecutar la siguiente fórmula obtenemos la misma fórmula como respuesta.

```
{log}=> Turn is 2*N + 1.

true
Constraint: Turn is 2*N+1, integer(Turn), integer(N)
```

Es decir, {*log*} nos dice que la fórmula es satisfacible pero no tenemos una de sus soluciones concretas. Si activamos CLP(FD) {*log*} muestra una advertencia que indica que la respuesta no es confiable.

```
{log}=> int_solver(clpfd).
{log}=> Turn is 2*N + 1.

***WARNING***: non-finite domain

true
Constraint: Turn is 2*N+1, integer(Turn), integer(N)
```

Esto indica que *posiblemente* la fórmula sea satisfacible pero CLP(FD) no puede asegurarlo. Si queremos una respuesta más segura tenemos que asociar `Turn` o `N` a un dominio finito:

[15] SWI-Prolog: CLP(FD).
[16] SWI-Prolog: CLP(Q).
[17] SWI-Prolog: Enumeration predicates.

```
{log}=> N in int(1,5) & Turn is 2*N + 1.

N = 1, Turn = 3
```

Interactivamente podemos obtener varias soluciones más. Por el contrario, si activamos CLP(Q) el dominio finito no sirve para encontrar una solución concreta:

```
{log}=> int_solver(clpq).
{log}=> N in int(1,5) & Turn is 2*N + 1.

true
Constraint: N>=1, N=<5, Turn is 2*N+1
```

La ventaja de CLP(Q) es que es completo para la aritmética lineal entera mientras CLP(FD) no lo es. Esto significa que si se quiere *demostrar automáticamente* una propiedad de un programa {*log*} que involucra aritmética entera lineal *para todos los números enteros* se debe usar CLP(Q)[18]. Si el programa involucra aritmética entera no-lineal, {*log*} será incapaz de demostrar propiedades. En ese caso lo más lejos que se puede llegar es activar CLP(FD) y usar dominios finitos para evaluar si al menos en esos dominios la propiedad vale[19].

Una forma general de evitar respuestas de {*log*} que contengan restricciones aritméticas enteras es activar CLP(Q) y `groundsol`.

```
{log}=> int_solver(clpq).
{log}=> groundsol.
{log}=> Turn is 2*N + 1.

Turn = 1, N = 0
```

Tener en cuenta que estas cuestiones sobre la ejecución de simulaciones que involucran aritmética entera aplican a las simulaciones simbólicas. El entorno Next se encarga de resolver estos problemas pero no puede ejecutar simulaciones simbólicas.

5. Demostraciones automáticas con {*log*}

Ejecutar simulaciones sobre una especificación {*log*} ayuda a ganar confianza en la corrección de la especificación. Sin embargo, sería mejor si pudiéramos *demostrar* que la especificación verifica ciertas propiedades (en el sentido lógico o matemático). A continuación veremos cómo {*log*} nos permite demostrar que un programa/especificación verifica ciertas propiedades.

Hasta el momento hemos usado {*log*} como un lenguaje de programación (de prototipos). Sin embargo, {*log*} también es un *satisfiability solver*, es decir es un programa que determina si las fórmulas de una cierta teoría son *satisfacibles* o no. En este caso la teoría es la de conjuntos y relaciones finitas combinada con el álgebra lineal entera[20].

[18] Como en general la aritmética no lineal es no decidible, difícilmente se pueda construir una herramienta que demuestre automáticamente propiedades de programas que incluyen aritmética no lineal.

[19] Si los dominios son muy grandes la demostración tomará un tiempo inaceptable. Existen algunas herramientas que implementan procedimientos de decisión o heurísticas para algunos fragmentos de la aritmética entera no-lineal.

[20] En lo que sigue hablaremos de la teoría de conjuntos finitos pero lo mismo vale para esta combinada con el álgebra lineal entera.

Recordemos que la fórmula F que depende de la variable x es satisfacible si y solo si:

$$\exists y : F(y)$$

En el caso de {*log*} la variable y se cuantifica sobre *todos* los conjuntos finitos. Por lo tanto, si {*log*} responde que F es satisfacible significa que existe (al menos) un conjunto finito que la satisface. Simétricamente, si {*log*} responde que F no es satisfacible significa que no hay ningún conjunto finito que la satisface, es decir:

$$\forall y : \neg F(y)$$

Ahora, si llamamos $G(x) \widehat{=} \neg F(x)$ y F no es satisfacible tenemos:

$$\forall y : G(y)$$

lo que significa que G es verdadera para todo conjunto finito. Dicho de otro modo, G es *válida* respecto de la teoría de conjuntos finitos; o, equivalentemente, G es un *teorema* de la teoría de conjuntos finitos.

Si {*log*} responde que *F es insatisfacible*, entonces sabemos que $\neg$ *F es un teorema*.

5.1. Lemas de invarianza

Una clase de propiedades importantes de las máquinas de estados son los *invariantes de estado* o, simplemente, *invariantes*. El predicado I, que depende de las variables de estado (y posiblemente de los parámetros de la especificación), es invariante respecto a la operación T si y solo si:

$$I \wedge T \Rightarrow I' \tag{1}$$

Las fórmulas como (1) se llaman *lemas de invarianza* y se dice que *T preserva I*.

Si queremos usar {*log*} para demostrar que (1) es un teorema tenemos que pedirle a {*log*} que determine si la negación de (1) es *insatisfacible*. O sea que en {*log*} tenemos que ejecutar:

$$\neg (I \wedge T \Rightarrow I') \tag{2}$$

Internamente {*log*} transforma (2) en:

$$I \wedge T \wedge \neg I' \tag{3}$$

En algunos casos {*log*} no va a ser capaz de calcular $\neg I'$. Cuando eso ocurra van a ver el mensaje `Unsafe use of negation for predicate...`. Si eso ocurre tienen que leer la sección 3.5 del manual de usuario de {*log*} y tratar de resolver el problema.

Si ven el mensaje anterior, leyeron la Sección 3.5 del manual y aun así tienen dudas de cómo resolver el problema, consulten con el docente.

Una de las causas más comunes por las que {*log*} no es capaz de calcular $\neg I'$ se debe a que el invariante contiene *variables existenciales*. Dado un predicado con signatura $p(x_1, \ldots, x_n)$, una variable es existencial en p si es una variable libre en el cuerpo de p distinta a todos los argumentos x_i de p. En otras palabras una variable es existencial cuando aparece libre en el cuerpo del predicado pero no aparece en su cabecera. Muchas variables existenciales que aparecen en los invariantes se pueden eliminar usando el predicado `let/3` (Sección 3.7) o un cuantificador existencial restringido (Sección 3.7.1).

Veamos un ejemplo donde `let/3` nos va a ayudar a evitar la introducción de variables existenciales en un invariante. Si el invariante de la agenda de cumpleaños fuese *known* $\subseteq$ dom *birthday*, en {*log*} tendríamos lo siguiente:

```
otroInvariante(Known,Birthday) :-
  dom(Birthday,Dom) & subset(Known,Dom).
```

{*log*} no puede calcular la negación de `otroInvariante` porque el predicado tiene una variable existencial, `Dom`. En este caso la solución a nuestro problema es usar `let/3`. En efecto, podemos escribir `otroInvariante` de esta forma:

```
otroInvariante(Known,Birthday) :-
  let([Dom], dom(Birthday,Dom), subset(Known,Dom)).
```

Ahora la variable `Dom` no está libre sino que está ligada al predicado `let/3`. Dicho de otro modo, `Dom` es una variable cuantificada existencialmente dentro del `let/3` por lo que no está libre en `otroInvariante`. {*log*} sabe cómo tratar ese tipo de variables pero no sabe hacerlo si la variable está libre. En consecuencia ahora {*log*} puede calcular la negación de `otroInvariante`.

Lean el apartado **The `let` predicate** de la Sección 3.5 del manual del usuario para aprender más sobre `let/3`.

Entonces, si ven el mensaje `Unsafe use of negation for predicate...` probablemente se deba a que tienen un invariante con variables existenciales. Si eso ocurre tienen que volver a escribir el invariante tratando de sacar esas variables mediante `let/3` o un cuantificador existencial restringido (ver el ejemplo de la Sección 6.1 del manual de usuario de {*log*}). Aun así pueden darse casos en que van a tener que escribir la negación manualmente. Si eso ocurre tienen que leer el apartado **Negation of user-defined predicates** de la Sección 3.5 del manual del usuario.

En el TP deben usar `let/3` siempre que sea posible (y correcto) para evitar problemas con la negación de invariantes. Caso contrario se les va a descontar puntos.

No usen `let/3` dentro de las operaciones porque trae problemas al generar casos de test (ver Sección 6).

Si `let/3` no se puede aplicar para alguno de los invariantes y tienen que escribir la negación a mano, pueden consultar con el docente.

5.2. El generador de condiciones de verificación (VCG)

El generador de condiciones de verificación (VCG) es un componente de {*log*} que automatiza la generación de lemas de invarianza y otras *condiciones de verificación* u *obligaciones de prueba* que son estándar en la verificación de máquinas de estado. Por ejemplo, además de los lemas de invarianza el VCG genera condiciones de verificación que demuestran que el estado inicial satisface cada uno de los invariantes. Estas condiciones de verificación se llaman *lemas de inicialización*. Si algún lema de inicialización no se verifica el sistema comenzaría a ejecutar en un estado que viola algún invariante y en consecuencia las operaciones no tendrían nada que preservar. En general demostrar un lema de inicialización es mucho más simple que demostrar los lemas de invarianza. Leer las Secciones 13.3 y 13.5 del manual del usuario de {*log*} para una descripción más completa del VCG.

Para trabajar con el VCG debemos primero consultar el archivo que contiene la especificación o programa y luego invocar el comando `vcg/1` con el mismo nombre de archivo. Para la especificación de la agenda de cumpleaños es así[21]:

```
{log}=> consult('bb.slog').
{log}=> vcg('bb.slog').
```

El VCG controla que se cumplan ciertas restricciones en la descripción de la máquina de estados tales como que haya una declaración `variables`, las declaraciones `invariant` se ubiquen antes de las declaraciones `operation`, etc. Si alguna de estas restricciones no se cumple se imprime el error correspondiente. El VCG también controla la presencia de *variables unitarias*[22]. Una variable unitaria es una variable que se usa solo una vez en el predicado. En general es un síntoma de error en el programa.

Si el comando `vcg` ejecuta con éxito se genera un archivo cuyo nombre es ⟨*arhivo*⟩`-vc.pl`, donde ⟨*arhivo*⟩ es el nombre del archivo que contiene la especificación. En el caso de la agenda de cumpleaños se genera el archivo `bb-vc.slog`. Este archivo contiene predicados que especifican las condiciones de verificación que deben ser *descargadas* (o sea que deben ser ejecutadas y demostradas). Primero tenemos que consultar ese archivo.

```
{log}=> consult('bb-vc.slog').
```

Para ejecutar las condiciones de verificación existe un comando con nombre `check_vcs_`⟨*arhivo*⟩; por ejemplo, `check_vcs_bb`.

```
{log}=> check_vcs_bb.

Checking birthdayBookInit_sat_birthdayBookInv ... OK
Checking birthdayBookInit_sat_pfunInv ... OK
```

[21] Ya hicimos esto cuando introdujimos el entorno Next.
[22] 'singleton variables', en inglés.

```
Checking addBirthday_is_sat ... OK
Checking findBirthday_is_sat ... OK
Checking remind_is_sat ... OK
Checking addBirthday_pi_birthdayBookInv ... OK
Checking addBirthday_pi_pfunInv ... ERROR
Checking findBirthday_pi_birthdayBookInv ... OK
Checking findBirthday_pi_pfunInv ... OK
Checking remind_pi_birthdayBookInv ... OK
Checking remind_pi_pfunInv ... OK
```

Las condiciones de verificación marcadas con **ERROR** deben ser examinadas porque indican un error en los invariantes, las operaciones o en algún otro elemento de la especificación.

Entregar el TP con condiciones de verificación marcadas con **ERROR** puede resultar en una calificación menor.

Por lo general el VCG va a marcar con error a algún lema de invarianza—es raro que las condiciones de verificación que no son lemas de invarianza den error. El siguiente cuadro resume los motivos por los que la demostración de un lema de invarianza puede fallar.

¿Por qué falla la demostración de un lema de invarianza?

En general cuando un lema de invarianza falla se debe a alguna de las siguientes razones:

1. El invariante es incorrecto
2. La operación tiene errores (faltan precondiciones, errores en postcondiciones, etc.)
3. Falta algún otro invariante (o un axioma) como hipótesis

Cuando el VCG no puede demostrar un lema de invarianza guarda un contraejemplo que suele ser muy útil para encontrar la causa del problema. Podemos ver el contraejemplo de la siguiente forma[23]:

```
{log}=> vcgce(addBirthday_pi_pfunInv).

Birthday = {[n2,n1]}
Known = {}
Name = n2
Date = n0
Known_ = {n2}
Birthday_ = {[n2,n0],[n2,n1]}
```

Observen que `Birthday = {[n2,n1]}` y `Known = {}`. O sea que el dominio de `Birthday` no es igual a `Known`. Pero al mismo tiempo el VGC ya demostró que `dom(Birthday,Known)` es

[23] El comando `vcgce` significa *verification condition ground counterexample* donde 'ground' refiere al comando `groundsol` de {*log*}.

un invariante de la especificación—ver las condiciones de verificación `*_pi_birthdayBookInv`. O sea que el contraejemplo se da cuando el estado de partida no verifica el *otro* invariante. Pero al mismo tiempo sabemos que los estados de partida verifican el otro invariante porque el VCG ya lo demostró. Este es el tercer tipo de problema señalado en ¿Por qué falla la demostración de un lema de invarianza?. Es decir falta un invariante cómo hipótesis del lema `addBirthday_pi_pfunInv`.

Para corregir este problema hacemos lo siguiente:

1. Editar el archivo `bb-vc.slog`.
2. Buscar la cláusula cuyo nombre es `addBirthday_pi_pfunInv`. Allí encontramos un comentario que dice:

   ```
   % here conjoin other ax/inv as hypothesis if necessary
   ```

3. Sustituir esa línea por:

   ```
   birthdayBookInv(Known,Birthday) &
   ```

   Notar que usamos los mismos nombres de variables que ya están presentes en la cláusula y que terminamos la línea con `&` (pues de lo contrario habría un error de sintaxis). De esta forma `addBirthday_pi_pfunInv` queda así:

   ```
   addBirthday_pi_pfunInv(
     Known, Birthday, Name, Date, Known_, Birthday_
   ) :-
       birthdayBookInv(Known,Birthday) &
       neg(
         pfunInv(Birthday) &
         addBirthday(Known,Birthday,Name,Date,Known_,Birthday_)
         implies pfunInv(Birthday_)
       ).
   ```

4. Grabar el archivo.
5. Consultar el archivo `bb-vc.slog` nuevamente.
6. Ejecutar el comando `check_vcs_bb` nuevamente.

Si el error no se debe a la falta de una hipótesis (o sea que se debe a alguna de las dos primeras causas antes explicadas) entonces vamos a tener que modificar la especificación. Por ejemplo, tendríamos que revisar el invariante para ver si no es demasiado fuerte, tendríamos que revisar la operación cuyo lema falla para ver si no falta una precondición o si una postcondición está mal. Luego de modificar la especificación tenemos que volver a ejecutar el comando `vcg`. En ese caso hay que tener en cuenta que el archivo con las condiciones de verificación se sobrescribe sin aviso por lo que si habíamos efectuado algún cambio (por ejemplo habíamos agregado alguna hipótesis) se perderá.

El comando `vcg` sobrescribe el archivo `*-vc.slog` sin aviso. Si agregaron hipótesis en ese archivo van a perder esas modificaciones.

El VCG también provee los comandos `vcace` y `findh` que se explican en la Secciones 13.4.1 y 13.4.2 del manual del usuario. `findh` es de alguna utilidad cuando el lema de invarianza no se puede probar porque faltan hipótesis.

Lemas de invarianza que dan *timeout*. Además de `OK` y `ERROR` el VCG puede marcar una condición de verificación con `TIMEOUT`. Esto significa que {*log*} intentó hacer la demostración durante un minuto pero no lo logró. Esto puede ocurrir cuando la demostración es muy compleja o cuando cae fuera de los procedimientos de decisión implementados en {*log*}. Esta situación rara vez ocurre con el tipo de problemas que se resuelven en el TP. Pero si ocurre, la solución la pueden encontrar al final de la Sección 13.3 del manual de usuario donde se presentan los comandos `check_vcs_*/1-2` para lo que seguramente también van a necesitar leer la Sección 11 del mismo manual.

Entregar el TP con condiciones de verificación marcadas con **`TIMEOUT`** puede resultar en una calificación menor. Si no pueden solucionar el problema, consulten con el docente.

6. Generación de casos de prueba con {*log*}-TTF

En esta sección vamos a explicar cómo usar la implementación del TTF[24] en {*log*}.

Asumimos que ya leyeron el material sobre el TTF.

{*log*}-TTF es una implementación reciente que ha tenido poco uso. Si ven comportamientos no esperados o tienen dudas sobre el funcionamiento, avisen al docente.

Antes de poder usar {*log*}-TTF se deben cumplir las siguientes condiciones:

1. La especificación ha pasado el control de tipos.
2. La especificación ha sido analizada por el VCG.
3. El archivo con las condiciones de verificación ha sido consultado.
4. No es necesario, aunque es muy recomendable, que se hayan descargado las condiciones de verificación generadas por el VCG.

Es decir se supone que ya hicieron lo siguiente.

[24] Para una referencia rápida pueden consultar Wikipedia.org: Test Template Framework.

```
{log}=> type_check.
{log}=> consult('bb.slog').
{log}=> vcg('bb.slog').
{log}=> consult('bb-vc.slog').
```

A continuación se debe inicializar {*log*}-TTF con este comando.

```
{log}=> ttf(⟨ atom ⟩).
```

donde *atom* puede ser cualquier átomo de Prolog[25]. En general este identificador debe ser algo relacionado con la especificación como por ejemplo el nombre del archivo. Por ejemplo, para la especificación de la agenda de cumpleaños podemos hacer:

```
{log}=> ttf(bb).
```

`bb` se va a usar para nombrar algunos archivos que genera {*log*}-TTF.

Ahora podemos empezar a aplicar tácticas de testing a alguna de las operaciones de la especificación. En {*log*} la primera táctica de testing que se aplica es Forma Normal Disyuntiva (DNF). Vamos a aplicar DNF a la operación `addBirthday` de la agenda de cumpleaños. El comando es el siguiente.

```
{log}=> applydnf(addBirthday(Name,Date)).
```

Como se ve, el comando espera un término de la forma $operation(i_1,\ldots,i_n)$ donde $i_1,\ldots,i_n$ son los parámetros de entrada de la operación, *sin incluir las variables de estado-anterior*. En otras palabras, el término $operation(i_1,\ldots,i_n)$ no debe mencionar variables de estado. {*log*}-TTF usa la estructura de la especificación para determinar si el término puede se aceptado o no. Los usuarios son los que deciden cuáles argumentos de la operación son de entrada y cuáles no. `applydnf` incluye automáticamente las variables de estado en el VIS de la operación.

Podemos ver el árbol de pruebas luego de aplicar una táctica con el siguiente comando (se usa indentación para mostrar los niveles del árbol).

```
{log}=> writett.

addBirthday_vis
    addBirthday_dnf_1
    addBirthday_dnf_2
```

Las clases de prueba se pueden exportar a un archivo con el siguiente comando[26].

```
{log}=> exporttt.
ttf: testing tree successfully exported to bb_addBirthday-tt.slog
```

El contenido de `bb_addBirthday-tt.slog` es básicamente el siguiente.

```
addBirthday_dnf_1(Name,Date,Known,Birthday) :- Name nin Known.
addBirthday_dnf_2(Name,Date,Known,Birthday) :- Name in Known.
```

[25] Un átomo de Prolog es cualquier secuencia de caracteres que empieza con una letra minúscula. Un átomo no se encierra entre comillas dobles.

[26] El prefijo `bb` en el nombre del archivo surge del comando `ttf/1`.

$S = \emptyset, T = \emptyset$	$S \neq \emptyset, T \neq \emptyset, S \subset T$
$S = \emptyset, T \neq \emptyset$	$S \neq \emptyset, T \neq \emptyset, T \subset S$
$S \neq \emptyset, T = \emptyset$	$S \neq \emptyset, T \neq \emptyset, T = S$
$S \neq \emptyset, T \neq \emptyset, S \cap T = \emptyset$	$S \neq \emptyset, T \neq \emptyset, S \cap T \neq \emptyset, S \not\subseteq T, T \not\subseteq S, S \neq T$

Figura 1: Partición estándar para $S \cup T$, $S \cap T$ y $S \setminus T$

Una vez que `applydnf` se aplicó a una operación de la especificación todas las tácticas que el usuario aplique de allí en más se aplicarán sobre la misma operación hasta que se vuelva a ejecutar el comando `ttf/1`, luego de lo cual `applydnf/1` se puede aplicar a otra (o la misma) operación. Cada llamada a `ttf/1` reinicia el proceso de generación de casos de prueba.

Luego de DNF cada táctica se aplica sobre uno o más nodos del árbol de pruebas. En este caso vamos a aplicar Particiones Estándar (SP) a la clase de prueba `addBirthday_dnf_1`. El comando es el siguiente.

```
applysp(addBirthday_dnf_1,un(Known,{Name},Known_)).
```

El primer argumento es el nodo a partir de cual se aplica la táctica y el segundo es un predicado atómico tal cual aparece en la operación—ver `addBirthdayOk` en la Sección 2.1.2. En la Figura 1 tienen la partición estándar para unión, intersección y diferencia por si no la recuerdan. En este caso la usamos para la unión `un(Known,{Name},Known_)`.

Los operadores soportados por SP en {*log*}-TTF y las particiones definidas para ellos se pueden explorar mirando el archivo `ttf_sp.pl` que se encuentra en el directorio de {*log*}. Los usuarios pueden extender SP agregando particiones para otros operadores disponibles en {*log*}. Observen el patrón de las particiones existentes y lean la escasa documentación del archivo. Comparen la Figura 1 con las cláusulas del predicado `ttf_sp/4` que tengan primer argumento `un`, `inters` o `diff`. Cada vez que modifiquen el archivo `ttf_sp.pl`, consúltenlo desde {*log*} para controlar mínimamente los cambios. Nuevas versiones de {*log*} van a proveer mejores formas de extender SP. De todas formas para el TP no deberían necesitar modificar ese archivo. Si lo modifican consulten con el docente.

El árbol de pruebas generado luego del comando `applysp` es el siguiente.

```
{log}=> writett.

addBirthday_vis
    addBirthday_dnf_1
        addBirthday_sp_11
        addBirthday_sp_12
        addBirthday_sp_13
        addBirthday_sp_14
        addBirthday_sp_15
        addBirthday_sp_16
        addBirthday_sp_17
```

```
        addBirthday_sp_18
    addBirthday_dnf_2
```

A continuación listamos dos clases de prueba como ejemplo—pueden exportar el árbol con el comando `exporttt`.

```
addBirthday_sp_11(Name,Date,Known,Birthday) :-
  Name nin Known & Known = {} & {Name} = {}.
addBirthday_sp_14(Name,Date,Known,Birthday) :-
  Name nin Known & Known neq {} &
  {Name} neq {} & disj(Known,{Name}).
```

Noten que `addBirthday_sp_11` es insatisfacible. Como vimos en clase esto es habitual en el TTF y esas clases de prueba deben ser eliminadas (*podadas*) del árbol de pruebas. {*log*}-TTF provee el comando `prunett`[27], que itera sobre las hojas del árbol de pruebas y le pregunta a {*log*} si son satisfacibles o no. Si {*log*} detecta que una hoja es insatisfacible `prunett` la elimina del árbol. El resultado de ejecutar `prunett` es el siguiente.

```
{log}=> prunett.
{log}=> writett.

addBirthday_vis
    addBirthday_dnf_1
        addBirthday_sp_12
        addBirthday_sp_14
    addBirthday_dnf_2
```

Ejecuten `prunett` cada vez que aplican una táctica desde la segunda en adelante.

Finalmente aplicamos una tercera táctica conocida como *cardinalidad de conjuntos* (SC). El comando es como el que sigue.

```
{log}=> applysc(addBirthday_dnf_1,Birthday).
```

El primer argumento es el nodo a partir del cual aplicamos la táctica y el segundo es una variable de tipo conjunto usada en la operación. El resultado es el siguiente.

```
{log}=> writett.

addBirthday_vis
  addBirthday_dnf_1
    addBirthday_sp_12
      addBirthday_sc_121
      addBirthday_sc_122
      addBirthday_sc_123
    addBirthday_sp_14
```

[27] `prunett` significa *prune testing tree*.

```
      addBirthday_sc_141
      addBirthday_sc_142
      addBirthday_sc_143
  addBirthday_dnf_2
```

Podríamos haber aplicado SC a, digamos, `addBirthday_sp_14` en cuyo caso solo esa clase de prueba habría sido particionada. De igual forma, la podríamos haber aplicado sobre la raíz del árbol (`addBirthday_vis`) en cuyo caso también `addBirthday_dnf_2` habría sido particionada. A qué nivel del árbol o a cuál nodo se aplica cada táctica depende del plan de testing y de lo que queremos testear.

SC particiona una clase de prueba en tres nuevas clases caracterizadas por los siguientes predicados: $V = \emptyset$, $V = \{X\}$ y $V = \{X, Y/W\} \wedge X \neq Y$, donde V es la variables pasada al comando `applysc`, y X, Y y W son variables nuevas. En otras palabras, SC asocia V con el conjunto vacío, con un conjunto unitario y con un conjunto con al menos dos elementos. El fundamento detrás de SC es que los conjuntos van a ser implementados con estructuras de datos que van a ser recorridas dentro de ciclos (*loops*). Luego, si un caso de prueba es con el conjunto vacío significa que el programa no va a entrar en ese ciclo; si un caso de prueba es con un conjunto unitario, el ciclo va a iterar solo una vez; y si un caso de prueba es con un conjunto con dos o más elementos, entonces el ciclo va a iterar más de una vez.varias veces. De esta forma el ciclo va a ser testeado con una cobertura razonable.

Como indicamos más arriba, ejecutamos `prunett` nuevamente aunque en este caso no poda ninguna hoja.

```
{log}=> prunett.
{log}=> writett.

addBirthday_vis
  addBirthday_dnf_1
    addBirthday_sp_12
      addBirthday_sc_121
      addBirthday_sc_122
      addBirthday_sc_123
    addBirthday_sp_14
      addBirthday_sc_141
      addBirthday_sc_142
      addBirthday_sc_143
  addBirthday_dnf_2
```

Como ya explicamos en clase, cuanto más profundo es el árbol de pruebas, más largos (complejos y restrictivos) son los predicados de las hojas. A su vez, cuanto más restrictivo es el predicado de una clase de prueba, más específico (o preciso o quirúrgico) es el caso de prueba que se va a generar a partir de ella. En consecuencia, un árbol de pruebas profundo significa, en general, una mejor cobertura de testing.

El paso final del proceso de testing es generar un caso de prueba a partir de cada hoja del árbol. {*log*}-TTF provee el comando `gentc`[28].

[28] `gentc` significa *generate test cases*.

```
{log}=> gentc.
{log}=> writett.

addBirthday_vis
  addBirthday_dnf_1
    addBirthday_sp_12
      addBirthday_sc_121 -> addBirthday_tc_121
      addBirthday_sc_122 -> addBirthday_tc_122
      addBirthday_sc_123 -> addBirthday_tc_123
    addBirthday_sp_14
      addBirthday_sc_141 -> addBirthday_tc_141
      addBirthday_sc_142 -> addBirthday_tc_142
      addBirthday_sc_143 -> addBirthday_tc_143
  addBirthday_dnf_2 -> addBirthday_tc_2
```

Cada nodo etiquetado con `_tc_` es un caso de prueba. Los casos de prueba se pueden explorar con el comando `writetc/1`.

```
{log}=> writetc(addBirthday_tc_123).

addBirthday_tc_123 is:
Name = name:n2,
Known = {},
Birthday = {[name:n0,date:n0],[name:n1,date:n1]}
```

El comando `writetc/0` imprime todos los casos de prueba con el formato mostrado más arriba, y `exporttt` exporta los casos de prueba y las clases de prueba correspondientes a un archivo.

Si queremos generar casos de prueba, por ejemplo, para la operación `remind` primero ejecutamos `ttf/1` nuevamente; luego `applydnf(remind(Today))` seguido por otras tácticas de testing (intercaladas con `prunett`) hasta que tengamos una cobertura razonable; finalmente invocamos `gentc`. Observen que `Today` es la única variable de entrada de `remind` dado que `Cards` debería ser el resultado de la operación.

Si `gentc` se ejecuta sobre un árbol de pruebas que contiene hojas insatisfacibles (porque en algún momento nos olvidamos de llamar a `prunett`), no se van a generar casos de prueba para esas hojas. En ese caso ejecuten `prunett` para podar las hojas insatisfacibles.

Entregar el TP con hojas del árbol de pruebas sin un caso de prueba puede resultar en una calificación menor. Si no pueden solucionar el problema, consulten con el docente.

La Tabla 11 del manual del usuario de {*log*} resume todas las tácticas de testing provistas por {*log*}. La Tabla 12 del mismo manual lista los comandos disponibles para trabajar con {*log*}-TTF.

Referencias

[1] Maximiliano Cristiá. Introducción a la notación Z. http://www.fceia.unr.edu.ar/asist/a-z.pdf, 2017. Apunte de clase - Ingeniería de Software 1.

[2] Maximiliano Cristiá and Gianfranco Rossi. A set solver for finite set relation algebra. In Jules Desharnais, Walter Guttmann, and Stef Joosten, editors, *Relational and Algebraic Methods in Computer Science - 17th International Conference, RAMiCS 2018, Groningen, The Netherlands, October 29 - November 1, 2018, Proceedings*, volume 11194 of *Lecture Notes in Computer Science*, pages 333–349. Springer, 2018.

[3] Maximiliano Cristiá and Gianfranco Rossi. Solving quantifier-free first-order constraints over finite sets and binary relations. *J. Autom. Reason.*, 64(2):295–330, 2020.

[4] Maximiliano Cristiá and Gianfranco Rossi. Automated proof of Bell-LaPadula security properties. *J. Autom. Reason.*, 65(4):463–478, 2021.

[5] Maximiliano Cristiá and Gianfranco Rossi. Automated reasoning with restricted intensional sets. *J. Autom. Reason.*, 65(6):809–890, 2021.

[6] Maximiliano Cristiá and Gianfranco Rossi. An automatically verified prototype of the Tokeneer ID station specification. *J. Autom. Reason.*, 65(8):1125–1151, 2021.

[7] Maximiliano Cristiá and Gianfranco Rossi. Integrating cardinality constraints into constraint logic programming with sets. *Theory Pract. Log. Program.*, 23(2):468–502, 2023.

[8] Maximiliano Cristiá and Gianfranco Rossi. A decision procedure for a theory of finite sets with finite integer intervals. *ACM Trans. Comput. Log.*, 25(1):3:1–3:34, 2024.

[9] Maximiliano Cristiá, Gianfranco Rossi, and Claudia S. Frydman. {*log*} as a test case generator for the Test Template Framework. In Robert M. Hierons, Mercedes G. Merayo, and Mario Bravetti, editors, *SEFM*, volume 8137 of *Lecture Notes in Computer Science*, pages 229–243. Springer, 2013.

[10] Agostino Dovier, Eugenio G. Omodeo, Enrico Pontelli, and Gianfranco Rossi. A language for programming in logic with finite sets. *J. Log. Program.*, 28(1):1–44, 1996.

[11] Agostino Dovier, Carla Piazza, Enrico Pontelli, and Gianfranco Rossi. Sets and constraint logic programming. *ACM Trans. Program. Lang. Syst.*, 22(5):861–931, 2000.

[12] J. M. Spivey. *The Z notation: a reference manual*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1992.

A. Código {*log*} de la agenda de cumpleaños

```
variables([Known,Birthday]).

def_type(bb,rel(name,date)).
def_type(kn,set(name)).

invariant(birthdayBookInv).
dec_p_type(birthdayBookInv(kn,bb)).
birthdayBookInv(Known,Birthday) :- dom(Birthday,Known).

invariant(pfunInv).
dec_p_type(pfunInv(bb)).
pfunInv(Birthday) :- pfun(Birthday).

initial(birthdayBookInit).
```

```
dec_p_type(birthdayBookInit(kn,bb)).
birthdayBookInit(Known,Birthday) :- Known = {} & Birthday = {}.

dec_p_type(addBirthdayOk(kn,bb,name,date,kn,bb)).
addBirthdayOk(Known,Birthday,Name,Date,Known_,Birthday_) :-
  Name nin Known &
  un(Known,{Name},Known_) &
  un(Birthday,{[Name,Date]},Birthday_).

dec_p_type(nameAlreadyExists(kn,name)).
nameAlreadyExists(Known,Name) :-
  Name in Known.

operation(addBirthday).
dec_p_type(addBirthday(kn,bb,name,date,kn,bb)).
addBirthday(Known,Birthday,Name,Date,Known_,Birthday_) :-
  addBirthdayOk(Known,Birthday,Name,Date,Known_,Birthday_)
  or
  nameAlreadyExists(Known,Name) &
  Known_ = Known & Birthday_ = Birthday.

dec_p_type(findBirthdayOk(kn,bb,name,date)).
findBirthdayOk(Known,Birthday,Name,Date) :-
  Name in Known &
  applyTo(Birthday,Name,Date).

dec_p_type(notAFriend(kn,name)).
notAFriend(Known,Name) :- Name nin Known.

operation(findBirthday).
dec_p_type(findBirthday(kn,bb,name,date)).
findBirthday(Known,Birthday,Name,Date) :-
  findBirthdayOk(Known,Birthday,Name,Date)
  or
  notAFriend(Known,Name).

operation(remind).
dec_p_type(remind(kn,bb,date,kn)).
remind(Known,Birthday,Today,Cards) :-
  rres(Birthday,{Today},M) & dec(M,bb) &
  dom(M,Cards).
```