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. Instalación de {*log*} **3**

2. Traducción de Z a {*log*} **3**

2.1. Un ejemplo de traducción . . . 3

2.1.1. La especificación Z . . . 3

2.1.2. El código {*log*} . . . 5

2.2. Traducción de pares ordenados . . . 10

2.3. Traducción de conjuntos . . . 10

2.3.1. Conjuntos extensionales — Introducción a la unificación conjuntista . . . 10

2.3.2. Producto cartesiano . . . 11

2.3.3. Intervalos de números enteros . . . 12

2.4. Traducción de la aplicación de función . . . 12

2.5. Traducción de expresiones aritméticas . . . 13

2.6. Traducción de los operadores de la teoría de conjuntos . . . 13

2.7. Traducción de listas . . . 15

2.8. Traducción de operadores lógicos . . . 16

2.8.1. Cuantificadores . . . 19

2.9. Traducción de definiciones axiomáticas . . . 19

3. Simulación de prototipos {*log*} **22**

3.1. Simulaciones simples . . . 23

3.2. Chequeo de tipos y simulaciones . . . 26

3.3. Simulaciones simbólicas . . . 27

3.4. Simulaciones inversas . . . 30

3.5. Evaluación de propiedades . . . 31

3.6. Simulaciones que involucran aritmética entera . . . 32

4. Demostraciones automáticas con {*log*} **33**

4.1. Lemas de invarianza y negación de invariantes . . . 34

4.2. Pruebas que involucran aritmética entera . . . 38

4.3. Chequeo de tipos y demostraciones . . . 39

4.4. El generador de condiciones de verificación (VCG) . . . 40

A. Código {*log*} de la agenda de cumpleaños **42**

1. Instalación de {*log*}

{*log*} ('setlog') es un lenguaje de programación basado en el paradigma de programación lógica de restricciones, 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 la extensión de {*log*} a relaciones binarias y otras teorías relacionadas.

{*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`

Recomendamos enfáticamente que antes de leer este documento estudien el manual de usuario de {*log*} (al menos los doce primeros capítulos):

`https://www.clpset.unipr.it/SETLOG/setlog-man.pdf`

Este apunte se focaliza en mostrar cómo traducir o implementar especificaciones Z usando el lenguaje de {*log*} y, luego, cómo efectuar simulaciones y demostraciones automáticas 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, 8, 9, 4, 6, 7].

2. Traducción de Z a {*log*}

Como la mayoría de las especificaciones Z tienen una estructura muy similar, comenzaremos mostrando cómo 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 12 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 cláusulas o predicados {*log*}. Una cláusula {*log*} es algo así como un procedimiento o subrutina. Cada cláusula puede recibir cero o más

parámetros. Cuando se traduce un esquema Z que representa una operación del sistema, los parámetros que recibe la cláusula 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 la cláusula `birthdayBookInv` es un invariante. Notar que si bien *BirthdayBookInv* comienza con mayúscula `birthdayBookInv` lo hace con minúscula porque las cláusulas {*log*} deben comenzar con minúscula. El predicado `dom(Birthday,Known)` se interpreta como dom *Birthday* = *Known*. En la sección 4 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 (∅).

La interfaz de la cláusula {*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 11 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,set([name,date])).
def_type(kn,set(name)).
```

donde `bb` es un identificador o sinónimo del tipo `set([name,date])`. En `set([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, `[name,date]` corresponde al tipo del producto cartesiano entre `name` y `date` (es decir, es el equivalente a $NAME \times DATE$ en Z). A su vez, `set([name,date])` corresponde al tipo de todos los conjuntos de tipo `[name,date]` (es decir, es el equivalente a $\mathbb{P}(NAME \times DATE)$ en Z). O sea que `set([name,date])` corresponde al tipo de las relaciones binarias entre `name` y `date` (i.e., $NAME \leftrightarrow DATE$ en Z).

Dado que tipos de la forma `set([`τ_1`,`τ_2`])` son muy usados en {*log*}, existe una forma más concisa de escribir estos tipos: `rel(`τ_1`,`τ_2`)`. Luego, podemos escribir la primera declaración `def_type` de más arriba como:

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

Internamente {*log*} reescribe `rel(name,date)` como `set([name,date])`, de manera tal que ambas expresiones son equivalentes.

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 de la cláusula:

```
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:

```
nombre_de_la_clausula(parametros)
```

A su vez, `parametros` es una lista que se corresponde uno a uno con los parámetros de la cláusula 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 establecer o 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 establecer o 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 establecer o 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 podríamos agregar `pfun(Birthday)` a las cláusulas donde usamos `Birthday`. `pfun` es un operador {*log*} que implementa la definición de función (parcial) como una subclase de los conjuntos de pares ordenados. Pero entonces, ¿por qué no dijimos `pfun(B)` en algún lado de `addBirthday`? Porque la idea es usar {*log*} para demostrar que `pfun(B)` es un invariante de estado de la especificación. De hecho si hubiéramos escrito el esquema *BirthdayBook* con declaraciones normalizadas resultaría lo siguiente:

BirthdayBook

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \leftrightarrow DATE$$
$$birthday \in NAME \nrightarrow DATE$$

donde $birthday \in NAME \nrightarrow DATE$ 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

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

BirthdayBookInv

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

donde es claro que deberíamos declarar que `pfun(Birthday)` es un invariante de estado de la especificación. La traducción a {*log*} es la siguiente[1]:

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

El segundo esquema de operación a traducir es *FindBirthday*. Como hicimos anteriormente, debemos comenzar por los esquemas que son invocados desde *FindBirthday*.

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

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

donde `Date` representa la variable Z *date*!; y `applyTo(Birthday,Name,Date)` significa *Birthday*(*Name*) = *Date*.

Ahora podemos ver la traducción de *FindBirthday*:

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

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

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,kn,bb)).
remind(Known,Birthday,Today,Cards,Known_,Birthday_) :-
  rres(Birthday,{Today},M) & dec(M,bb) &
  dom(M,Cards) &
  Known_ = Known &
  Birthday_ = Birthday.
```

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`".

2.2. 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 2.5). 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.

2.3. Traducción de conjuntos

2.3.1. 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 $\setminus$ (ver Tabla 1):

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

Pero también se puede traducir usando un conjunto extensional:

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

lo que en general será más eficiente.

Es decir el predicado `A = {D_i / A_}` *unifica* `A` con `{D_i / 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_i nin A_`? Simplemente porque, por ejemplo, `A = {1,2}`, `D_i = 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_i nin A_` eliminamos todas las soluciones donde `D_i` pertenece a `A_`.

{*log*} resuelve las igualdades de la forma `B = C`, donde `B` y `C` son términos que denotan conjuntos, usando *unificación conjuntista*[2]. La unificación conjuntista está en la base del poder deductivo y de cómputo de {*log*} constituyendo una extensión importante del algoritmo de 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.

2.3.2. 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 $A \times B$.

[2] 'set unification' en inglés.

2.3.3. 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 2.5 para la traducción de expresiones artiméticas):

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

2.4. 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. . . ".

Observen que si en una especificación escribimos `pfun(F)` le estamos indicando al programador que debe controlar que la estructura de datos sea una función cada vez que quiera aplicar la función a un argumento. Lo más razonable es que el código sea tal que garantice que la estructura de datos sea una función sin tener que controlarlo todo el tiempo. Por este motivo las formas correctas de expresar la aplicación de función son las que se muestran más arriba sumado a demostrar que `pfun(F)` es un invariante.

La definición de `applyTo` es la siguiente:

```
applyTo(F,X,Y) :- F = {[X,Y] / G} & [X,Y] nin G & comp({[X,X]},G,{})
```

La justificación es la siguiente. Si sabemos que $x \in \mathrm{dom} f$ entonces existen `Y` y `G` tales que `F = {[X,Y] / G} & [X,Y] nin 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,{})`. 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) & X nin 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]`.

2.5. 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 la cláusula. El problema es que ni {*log*} ni Prolog evalúan expresiones aritméticas a menos que el programador lo pida usando el operador `is`. Es decir si en {*log*} ejecutamos `{X,Y - 4} = {Y - 3 - 1,X}`, la respuesta será `no` porque {*log*} intentará determinar si `Y - 4 = Y - 3 - 1` sin evaluar ambas expresiones (es decir las considerará, básicamente, como cadenas de caracteres donde `Y` es una variable entera y por lo tanto la igualdad es imposible sin importar el valor de `Y`). Por el contrario, si ejecutamos `{X,A} = {B,X} & A is Y - 4 & B is Y - 3 - 1` {*log*} retornará varias soluciones (algunas de ellas repetidas), lo que significa que los conjuntos son iguales de varias formas posibles.

Lo mismo aplica al predicado `neq`: para {*log*} `Y - 4 neq Y - 3 - 1` es verdadero. En consecuencia debemos escribir: `H is Y - 4 & U is Y - 3 - 1 & H neq U`. Sin embargo, con los operadores de orden no hay problema: `X + 1 > X` es satisfacible pero `X - 1 > X` no lo es.

Para indicar que el conjunto `A` es un subconjunto de los naturales se debe hacer mediante una cuantificación universal restringida (ver Sección 2.8.1):

```
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]).

2.6. 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 3.

Operador	{*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)`	$\lvert A \rvert = n$
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)`	$A \not\parallel B$

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

Operador	{*log*}	Significado
relación binaria	`rel(R)`	R es una relación binaria
función parcial	`pfun(R)`	R es una función parcial
aplicación de función	`apply(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.		

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

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 la cláusula) 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.

2.7. 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 un fragmento de la teoría de listas que es decidible. Vamos a ver cómo usar ese fragmento.

En primer lugar no vamos a traducir a listas sino que vamos a utilizar una estructura de datos similar que, además de ordenar los elementos y permitir repeticiones, mantiene la longitud de forma explícita. En {*log*} esta estructura de datos se llama arreglo (*array)*. Para poder trabajar con arreglos es necesario cargar el archivo `array.slog` (i.e. `add_lib('array.slog')`)[3].

Como se puede observar en la Tabla 3, un arreglo se declara mediante el predicado `arr/2` cuya definición es la siguiente[4]:

```
arr(A,N) :- 0 < N & pfun(A) & dom(A,int(1,N)).
```

Es decir que un arreglo de longitud n es una función parcial cuyo dominio es el intervalo $[1,n]$. O sea que un arreglo es un conjunto de pares ordenados tales que las primeras componentes forman el intervalo $[1,n]$. Pueden ver estos ejemplos de uso de `arr/2` para ver cómo funciona.

```
{log}=> arr({[1,a],[2,X]},N).

N = 2

{log}=> arr({[1,a],[5,X]},N).

no

{log}=> arr({[1,a],[I,X]},N).

I = 2,  N = 2

I = 1,  X = a,  N = 1

{log}=> arr(A,N) & [I,X] in A & [I,Y] in A & X neq Y.

no
```

[3]Descargar el archivo en http://www.fceia.unr.edu.ar/is2/array.slog.
[4]Pueden ver el código abriendo el archivo `array.slog`.

```
{log}=> neg(arr(A,N) & arr(A,M) implies N = M).

no
```

Tener en cuenta que si bien `arr/2` requiere indicar la longitud del arreglo, si esta permanece variable entonces el arreglo puede almacenar un número finito, pero arbitrario, de elementos lo que lo hace muy parecido a una lista.

Para usar los operadores de la Tabla 3 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)`, `B` debe ser un arreglo de longitud `n+1`.
- En `prefix(A,n,k,B)`, `B` debe ser un arreglo de longitud `k`.
- En los operadores denominados "pseudo", `B` es una función pero no necesariamente es un arreglo. En muchos casos se pierde la propiedad de que el dominio de `B` 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 3, consulten con el docente.

2.8. 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[5] del usuario de {*log*} para la lista completa y otras consideraciones importantes).

[5] https://www.clpset.unipr.it/SETLOG/manual_4_9_8.pdf

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,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)`	$B = A \rhd S$

Cuadro 3: Operadores de arreglos disponibles en {*log*}

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 $\neg$ `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 $A \nsubseteq B$ 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.

Usamos `neg` para traducir este esquema Z:

[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.

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

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 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. Por este motivo {*log*} 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`.

2.8.1. 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.

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; y también están disponibles los cuantificadores existenciales restringidos (REQ)[8].

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

2.9. 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`. Todas las cláusulas declaradas 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.

[8] 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.

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.

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 en una cláusula.

```
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

$$checksum : \mathrm{seq}\, BYTE \rightarrow BYTE$$

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.
```

3. 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[9]. 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).

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

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.

3.1. Simulaciones simples

Veamos un ejemplo de simulación de un prototipo {*log*}. Consideremos el prototipo de la agenda de cumpleaños asumiendo que está almacenado en el archivo `bb.pl`. Empezamos ejecutando el intérprete Prolog desde una terminal de comandos y desde el directorio donde instalamos {*log*}[10].

```
~/setlog$ prolog

?- consult('setlog.pl').

?- setlog.

{log}=> consult('bb.pl').

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

Another solution?  (y/n) y
no
{log}=>
```

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. Efectuamos la simulación:

   `birthdayBookInit(K,B) & addBirthday(K,B,maxi,160367,K_,B_).`

   la cual consiste en:

   - Invocar la cláusula `birthdayBookInit` pasándole como parámetros cualesquiera variables;
   - Invocar la cláusula `addBirthday` pasándole en los dos primeros parámetros las mismas variables que usamos en la invocación a `birthdayBookInit`; como tercer y cuarto parámetro dos constantes y en los dos últimos parámetros dos variable nuevas.

[10] El nombre del ejecutable del intérprete Prolog puede variar dependiendo del sistema operativo.

Notar que la simulación termina con un punto.

6. {*log*} muestra el resultado de la simulación
7. {*log*} pregunta si queremos ver otras posibles soluciones a lo que respondemos que sí.
8. {*log*} informa que no hay más soluciones.

Veamos con más detalle la simulación:

```
birthdayBookInit(K,B) & addBirthday(K,B,maxi,160367,K_,B_).
```

Cuando invocamos `birthdayBookInit(K,B)` se produce la unificación de `K` y `B` con `Known` y `Birthday` que son los parámetros formales usados en la definición de `birthdayBookInit` (ver el código completo en el Apéndice A). Esto implica que `K` es igual a `Known` y `B` lo es a `Birthday` lo que a su vez implica que `K` y `B` son iguales a `{}`, que es lo que indica la primera línea de la respuesta de {*log*}. Entonces, cuando invocamos `addBirthday(K,B,maxi,160367,K_,B_)` es como si invocáramos:

```
addBirthday({},{},maxi,160367,K_,B_)
```

La invocación a `addBirthday` implica la invocación *no determinista* de `addBirthdayOk` y `nameAlready Exists`. Es decir que ambas cláusulas se invocan en un orden indeterminado. Supongamos que se invoca primero a `addBirthdayOk`. En este caso, se producen las siguientes unificaciones:

```
Known = {}
Birthday = {}
Name = maxi
Date = 160367
K_ = Known_
B_ = Birthday_
```

Por lo tanto el predicado {*log*} de `addBirthdayOk` queda de la siguiente forma:

```
maxi nin {} &
un({},{maxi},K_) &
un({},{[maxi,160367]},B_)
```

lo cual reduce a lo siguiente:

```
K_ = {maxi} &
B_ = {[maxi,160367]}
```

que corresponde a la segunda línea de la respuesta de {*log*}.

Al pulsar 'y' cuando {*log*} pregunta si queremos otra solución se invoca la cláusula `nameAlready Exists` porque es la cláusula que estaba pendiente de invocación. Nuevamente se produce una serie de unificaciones:

```
Known = {}
Birthday = {}
Name = maxi
K_ = Known
B_ = Birthday
```

Entonces el predicado {*log*} de `nameAlreadyExists` queda de la siguiente forma:

```
maxi in {}
```

Como este predicado es evidentemente falso, significa que la invocación de `nameAlreadyExists` falla y por lo tanto no produce ningún resultado. De aquí que {*log*} responda `no` cuando le pedimos otra solución.

La siguiente es una simulación más larga que incluye las dos invocaciones de la simulación que acabamos de analizar en detalle.

```
birthdayBookInit(K,B)                  & addBirthday(K,B,maxi,160367,K1,B1) &
addBirthday(K1,B1,'Yo',201166,K2,B2)   & findBirthday(K2,B2,'Yo',C,K3,B3) &
addBirthday(K3,B3,'Otro',201166,K4,B4) & remind(K4,B4,160367,Card,K5,B5) &
remind(K5,B5,201166,Card1,K_,B_).
```

donde podemos observar que invocamos todas las operaciones definidas; que usamos diferentes variables para encadenar los estados y capturar las salidas como `Card` y `Card1`; y que es posible ingresar constantes que empiezan con mayúscula siempre y cuando las encerremos entre comillas simples.

La solución que retorna la simulación anterior es:

```
K = {},
B = {},
K1 = {maxi},
B1 = {[maxi,160367]},
K2 = {maxi,Yo},
B2 = {[maxi,160367],[Yo,201166]},
C = 201166,
K3 = {maxi,Yo},
B3 = {[maxi,160367],[Yo,201166]},
K4 = {maxi,Yo,Otro},
B4 = {[maxi,160367],[Yo,201166],[Otro,201166]},
Card = {maxi},
K5 = {maxi,Yo,Otro},
B5 = {[maxi,160367],[Yo,201166],[Otro,201166]},
Card1 = {Yo,Otro},
K_ = {maxi,Yo,Otro},
B_ = {[maxi,160367],[Yo,201166],[Otro,201166]}
```

donde podemos notar que {*log*} nos brinda la posibilidad de contar con una traza completa de la ejecución del prototipo. Observar también que {*log*} internamente elimina las comillas simples que usamos para algunas constantes.

Una simulación donde siempre se usara las mismas variables de estado, por ejemplo:

```
birthdayBookInit(K,B) & addBirthday(K,B,N,C,K,B).
```

en general fallaría porque sería imposible que los valores de `K` y `B` antes de invocar a `addBirthday` unificaran con los que se obtienen cuando esa cláusula termina. Podríamos usar la misma variable para el estado anterior y posterior cuando la cláusula corresponde a una operación Z que no modifica el estado (por ejemplo cuando invocamos `findBirthday` y `remid`).

Hasta el momento las dos simulaciones que realizamos las hicimos desde el estado inicial. Es muy simple iniciar una simulación desde un estado cualquiera:

```
K = {maxi,caro,cami,alvaro} &
B = {[maxi,160367],[caro,201166],[cami,290697],[alvaro,110400]} &
addBirthday(K,B,'Yo',160367,K1,B1) & remind(K1,B1,160367,Card,K1,B1).
```

donde podemos ver que usamos la misma variable para indicar el estado anterior y final de `remid` (porque sabemos que no produce un cambio de estado). En este caso la salida es:

```
K = {maxi,caro,cami,alvaro},
B = {[maxi,160367],[caro,201166],[cami,290697],[alvaro,110400]},
K1 = {maxi,caro,cami,alvaro,Yo},
B1 = {[maxi,160367],[caro,201166],[cami,290697],[alvaro,110400],[Yo,160367]},
Card = {maxi,Yo}
```

Un problema que puede presentarse cuando se define manualmente el estado inicial para una simulación es que este, por error humano, no sea consistente con, por ejemplo, el invariante de estado. No obstante es muy simple evitar este problema como veremos en la sección 3.5.

3.2. Chequeo de tipos y simulaciones

En todo lo que hicimos en la sección anterior {*log*} jamás usó 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.pl`. Es decir que si había algún error de tipos 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 11 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$ prolog

?- consult('setlog.pl').

?- setlog.

{log}=> type_check.         % activamos el type checker

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

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:maxi,date:160367,K_,B_) &
        dec([K,K_],kn) & dec([B,B_],bb).

K = {},
B = {},
K_ = {name:maxi},
B_ = {[name:maxi,date:160367]}
```

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[11].

Si uno quiere chequear los tipos de un programa, por ejemplo de `bb.pl`, 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.

En lo que sigue de esta sección trabajaremos con simulaciones no tipadas. Esto significa que para ejecutar estas simulaciones el usuario debe asegurarse de que esté desactivado el control de tipos (comando `notype_check`).

3.3. Simulaciones simbólicas

Ejecutar 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. Las condiciones para que {*log*} pueda realizar simulaciones simbólicas son las siguientes[12]:

1. No debe haber definiciones recursivas.
2. Solo se usan los operadores de las Tablas 1 y 2. Si el programa {*log*} usa cardinalidad (`size`) no debe usar los operadores de la Tabla 2. El operador de cardinalidad es completo solo en combinación con los operadores de la Tabla 1.

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

[12] 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.

3. Todas las fórmulas aritméticas son lineales[13].

Es decir que el código {*log*} del prototipo no puede incluir operadores de la Tabla 3 si se quiere realizar simulaciones simbólicas[14]. En realidad, muchas simulaciones simbólicas son aun posible aunque las condiciones de arriba no se verifiquen.

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:

```
birthdayBookInit(K,B) & addBirthday(K,B,N,C,K_,B_).
```

ante lo cual {*log*} responde con:

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

que es una representación del resultado esperado.

Notar que la respuesta contiene variables a la derecha de las igualdades (`N` y `C`). Si preferimos ver constantes en lugar de variables 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 11.7 del manual de {*log*}.

Ahora desactivamos `groundsol` y encadenamos otra invocación más a `addBirthday` usando otro juego de variables para las entradas:

```
birthdayBookInit(K,B) & addBirthday(K,B,N,C,K1,B1) & addBirthday(K1,B1,M,D,K_,B_).
```

en cuyo caso la primera solución de {*log*} es:

```
K = {},
B = {},
```

[13]Más precisamente, las expresiones enteras son sumas o restas de términos de la forma `x*y` con `x` o `y` constante. Todos los operadores de orden están permitidos, incluso `neq`.

[14]El problema con los operadores de la Tabla 3 es que dependen de ciertos aspectos de la teoría de conjuntos que aun no han sido incorporados a {*log*}.

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

En este caso vemos una parte de la respuesta de {*log*} que no habíamos visto en las simulaciones anteriores (i.e. `Constraint`). En efecto, la solución más general que puede retornar {*log*} consta de dos partes: una lista (posiblemente vacía) de igualdades entre variables y términos (o expresiones); y una lista (posiblemente vacía) de *restricciones* (los predicados luego de la palabra `Constraint`). Cada restricción es un predicado {*log*}. La conjunción de todas las restricciones es siempre satisfacible (en general la solución se obtiene sustituyendo las variables de tipo conjunto por el conjunto vacío). 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:

```
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[15].

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*}.

3.4. Simulaciones inversas

Normalmente en una simulación el usuario provee las entradas y el modelo devuelve las salidas. En ocasiones resulta interesante tener la posibilidad de obtener las entradas partiendo de las salidas. Esto implica una suerte de simulación inversa.

{*log*} es capaz de realizar simulaciones inversas dentro de los mismos límites en los que es capaz de realizar simulaciones simbólicas. En realidad, mirando con atención el contenido de la sección anterior, resulta bastante claro que {*log*} no distingue entre variables de entrada y de salida ni entre estado anterior y siguiente. Por lo tanto, para {*log*} simular un prototipo dando valores a las variables de entrada o dándolos a las variables de salida no es muy distinto; de hecho {*log*} es capaz de computar solo con variables.

Veamos una simulación inversa muy simple, donde solo damos el valor para el estado siguiente:

```
K_ = {maxi,caro,cami,alvaro} &
B_ = {[maxi,160367],[caro,201166],[cami,290697],[alvaro,110400]} &
```

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

```
addBirthday(K,B,N,C,K_,B_).
```

donde la primera solución que retorna {*log*} es la siguiente:

```
K_ = {maxi,caro,cami,alvaro},
B_ = {[maxi,160367],[caro,201166],[cami,290697],[alvaro,110400]},
K = {maxi,caro,cami},
B = {[maxi,160367],[caro,201166],[cami,290697]},
N = alvaro,
C = 110400
```

Cuando la especificación Z es determinista, el prototipo {*log*} también lo será y por lo tanto para una entrada determinada habrá solo una solución. Sin embargo, aun con prototipos deterministas, una simulación inversa puede generar un gran número de soluciones. Este es el caso de la simulación anterior. En efecto, en la primera solución {*log*} consideró el caso donde `N = alvaro` y `C = 110400`, pero esta no es la única posibilidad. Entonces, cuando le solicitamos a {*log*} más soluciones obtenemos, por ejemplo:

```
K_ = {maxi,caro,cami,alvaro},
B_ = {[maxi,160367],[caro,201166],[cami,290697],[alvaro,110400]},
K = {maxi,caro,alvaro},
B = {[maxi,160367],[caro,201166],[alvaro,110400]},
N = cami,
C = 290697
```

lo que significa que para obtener `K_` y `B_` podemos partir de `K` y `B` donde no está el cumpleaños de `cami` y agregarlo.

3.5. Evaluación de propiedades

Al final de la sección 3.1 mostramos cómo iniciar una simulación desde un estado distinto del inicial. También dijimos que esto tiene cierto riesgo porque al escribir manualmente el estado inicial el usuario puede cometer errores creando un estado inconsistente. En esta sección veremos cómo evitar ese problema usando una característica de {*log*} que es útil para otras actividades de verificación.

Consideremos el siguiente estado de la agenda de cumpleaños:

```
Known = {maxi,caro,cami,alvaro}
Birthday = {[maxi,160367],[caro,201166],[cami,290697],[alvaro,110400]}
```

Iniciar una simulación desde este estado puede dar resultados incorrectos si no verifica el invariante de estado definido para la especificación. Recordemos que el invariante es:

BirthdayBookInv

$$BirthdayBook$$

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

Con {*log*} es muy simple comprobar que el estado definido arriba verifica el invariante. Primero traducimos el invariante. Simplemente invocamos `birthdayBookInv` pasándole el estado de más arriba como parámetro:

```
Known = {maxi,caro,cami,alvaro} &
Birthday = {[maxi,160367],[caro,201166],[cami,290697],[alvaro,110400]} &
birthdayBookInv(Known,Birthday).
```

En este caso {*log*} responde con los valores de `Known` y `Birthday`, lo que significa que la propiedad se cumple porque de lo contrario la respuesta hubiese sido `no`. Si invocamos `birthdayBookInv` pasándole directamente los valores de las variables:

```
birthdayBookInv({maxi,caro,cami,alvaro},
                {[maxi,160367],[caro,201166],[cami,290697],[alvaro,110400]}).
```

la respuesta es `yes`. Si por error hubiésemos escrito un estado que no verifica el invariante, por ejemplo (donde falta `maxi` en `Known`):

```
Known = {caro,cami,alvaro} &
Birthday = {[maxi,160367],[caro,201166],[cami,290697],[alvaro,110400]} &
birthdayBookInv(Known,Birthday).
```

la respuesta de {*log*} sería `no`.

En general podemos tomar cualquier propiedad que se nos ocurra y evaluarla para ciertos valores de sus variables. Por ejemplo, la componente `Birthday`, en cualquier estado, debe ser una función parcial. Entonces, por ejemplo, ante:

```
pfun({[maxi,160367],[caro,201166],[cami,290697],[alvaro,110400]}).
```

{*log*} responde `yes` y ante:

```
pfun({[maxi,160367],[maxi,201166],[cami,290697],[alvaro,110400]}).
```

responde `no`.

3.6. Simulaciones 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)[16] y CLP(Q)[17]. 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*[18]) 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:

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

```
Turn is 2*N + 1.
```

es exactamente la misma fórmula. Es decir, {*log*} nos dice que la fórmula es satisfacible pero no tenemos una de sus soluciones concretas. Si activamos CLP(FD):

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

{*log*} nos devuelve una advertencia y la misma fórmula:

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

true
Constraint: Turn is 2*N+1
```

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:

```
N in int(1,5) & Turn is 2*N + 1.
```

en cuyo caso la primera solución es:

```
N = 1, Turn = 3
```

e 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*} *para todos los números enteros* se debe usar CLP(Q)[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
```

4. Demostraciones automáticas con {*log*}

Evaluar propiedades con {*log*} ayuda a realizar simulaciones correctas y a comprobar que la especificación misma verifica esas propiedades. Sin embargo, sería mejor si pudiéramos

[19]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.

demostrar que la especificación verifica esas propiedades (en el sentido lógico o matemático). A continuación veremos cómo {*log*} nos permite demostrar que un programa 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 donde se usan los operadores de las Tablas 1 y 2 combinada con el álgebra lineal entera[20].

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 $G(x) \mathrel{\widehat{=}} \neg F(x)$ 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*.

4.1. Lemas de invarianza y negación de invariantes

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 sí y solo sí:

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

A las fórmulas como (1) se las llama *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 & T implies I')
```
(2)

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

Un inconveniente en usar {*log*} para ver si fórmulas como (2) son insatisfacibles, es que `neg` no siempre funciona correctamente, como explicamos en la Sección 2.8. En este caso particular tenemos que ver si `neg(I')` funciona bien porque: $\neg (I \wedge T \Rightarrow I') \equiv \neg (\neg (I \wedge T) \vee I') \equiv I \wedge T \wedge \neg I'$. Como pueden ver `neg` no tiene influencia sobre `I` ni `T`; en definitiva, solo se aplica sobre `I'`.

Como vimos en la Sección 2.1.2 los invariantes los introducimos en cláusulas declaradas con `invariant`. Un problema que tenemos es que {*log*} no calcula de forma automática la negación de cláusulas definidas por el usuario. Entonces, para demostrar lemas de invarianza necesitamos escribir la negación de cada invariante. En el programa {*log*} de la agenda de cumpleaños propusimos como invariante:

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

Entonces, antes de demostrar un lema de invarianza, tenemos que introducir la negación de `birthdayBookInv` de la siguiente forma:

```
n_birthdayBookInv(Known,Birthday) :- neg(dom(Birthday,Known)).
```

Esta nueva cláusula *no* va precedida de una declaración `invariant` pero *sí* se debe preceder de una declaración `dec_p_type` (ver el código completo en el Apéndice A). O sea que cuando queremos introducir la negación de un invariante tenemos que introducir una nueva cláusula cuyo nombre es de la forma `n_`⟨*nombreClausulaInvariante*⟩, con los mismos parámetros del invariante y cuyo cuerpo debe ser la negación del cuerpo del invariante.

En muchos casos la negación del cuerpo del invariante se consigue simplemente como en el ejemplo de arriba, es decir escribiendo `neg(`⟨*cuerpoInvariante*⟩`)`. Pero como `neg` no siempre funciona correctamente, puede haber casos donde es necesario calcular la negación manualmente. Veamos un ejemplo donde `let` (Sección 2.8) nos va a ayudar a calcular la negación. Si el invariante de la agenda de cumpleaños fuese $known \subseteq \mathrm{dom}\, birthday$, en {*log*} tendríamos lo siguiente:

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

Para calcular la negación de este invariante *no* podemos usar `neg` directamente porque tenemos una variable existencial, `Dom`. Es decir si escribimos:

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

{*log*} no será capaz de demostrar lemas de invarianza que involucren a `otroInvariante`. ¿Cuál es el resultado de `neg` en ese caso? ¿Por qué ese resultado no es el predicado que necesitamos? Les dejo estas preguntas como ejercicios. En este caso la solución a nuestro problema es usar `let`. En efecto, `otroInvariante` lo podemos escribir de esta forma:

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

Ahora podemos usar `neg` directamente:

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

¿Por qué ahora `neg` funciona bien?

Entonces, antes de escribir manualmente la negación de un invariante, veamos si usando `let` no podemos 'esconder' variables existenciales para lograr que `neg` funcione bien. Aun así habrá casos en que debamos escribir la negación manualmente.

> En el TP deben usar `let` siempre que sea posible (y correcto) para evitar escribir manualmente la negación de invariantes. Caso contrario les voy a descontar puntos.

Volvamos a los lemas de invarianza (fórmula (2)). Para demostrar con {*log*} que `addBirthday` preserva `birthdayBookInv`, tenemos que ejecutar:

```
neg(
  birthdayBookInv(K,B) & addBirthday(K,B,N,C,K_,B_)
  implies birthdayBookInv(K_,B_)
).
```

Si `addBirthday` preserva el invariante, la respuesta debería ser `no`; caso contrario deberíamos ver un contraejemplo. Si la respuesta es `no` significa que la fórmula es insatisfacible y por lo tanto la fórmula al interior de `neg` es un teorema.

Si al ejecutar el código de arriba vemos un mensaje del tipo `Unsafe use of negation for predicate` significa que no hemos incluido en el programa la negación de `birthdayBookInv`.

En definitiva hemos demostrado que `addBirthday` preserva `birthdayBookInv` para todo *conjunto finito*. Esto no es exáctamente lo mismo que hubiéramos demostrado usando, por ejemplo, Z/EVES pues en ese caso lo hubiéramos demostrado para todo *conjunto* (y no solo para los *finitos*). La ventaja de hacerlo con {*log*} es que las demostraciones son automáticas. Por otro lado, el programa que implemente la agenda de cumpleaños jamás operará con un conjunto infinito de gente conocida por lo que la demostración hecha con {*log*} resulta apropiada. Dejamos como ejercicio demostrar que las otras dos operaciones de la especificación preservan ese invariante.

Cuando tradujimos a {*log*} la especificación Z de la agenda de cumpleaños, mencionamos que al traducir *birthday* su tipo a nivel de {*log*} es `rel(name,date)`, el cual es equivalente al tipo Z $NAME \leftrightarrow DATE$. Además dijimos que el hecho de que *birthday* sea una función no lo codificamos como un tipo porque, en {*log*}, las funciones no son un tipo (como lo son en los lenguajes funcionales, por ejemplo). Dijimos que íbamos a usar {*log*} para demostrar que `pfun(Birthday)` es un invariante de estado. Es decir, con el sistema de tipos de {*log*} podemos expresar que `Birthday` es una relación binaria entre `name` y `date`; y luego podemos usar {*log*} para demostrar que en realidad `Birthday` es una función.

Recordar que el invariante `pfun(Birthday)` lo escribimos dentro de un predicado de usuario:

```
pfunInv(Birthday) :- pfun(Birthday).
```

por lo que tenemos que introducir la negación de ese predicado, como explicamos más arriba (lo dejo como ejercicio). Entonces para demostrar que `addBirthday` preserva `pfunInv` tenemos que ejecutar:

```
neg(pfunInv(B) & addBirthday(K,B,N,C,K_,B_) implies pfunInv(B_)).
```

En este caso {*log*} responde, tal vez inesperadamente, con una solución (en lugar de responder `no`). Es decir, {*log*} nos está diciendo que la fórmula que le dimos es satisfacible, o sea que no

es insatisfacible, o sea que la negación de esa fórmula *no* es un teorema. ¿Por qué no es un teorema cuando nosotros estamos bastante seguros de que debería serlo? La respuesta que nos da {*log*} nos debería ayudar a entender el problema. En este caso la respuesta de {*log*} es un *contraejemplo* porque es un ejemplo que contradice lo que nosotros creíamos que era un teorema. El contraejemplo es el siguiente:

```
B = {[N,_N2]/_N1},
K_ = {N/K},
B_ = {[N,_N2],[N,C]/_N1}
Constraint: pfun(_N1), comppf({[N,N]},_N1,{}), N nin K, set(_N1), [N,C] nin _N1,
            C neq _N2, [N,_N2]nin _N1, set(K)
```

Seguramente la respuesta les parece muy compleja. La podemos simplificar activando `groundsol` y ejecutando de nuevo:

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

B = {[n2,n1]},
K = {},
N = n2,
C = n0,
K_ = {n2},
B_ = {[n2,n0],[n2,n1]}
```

Observen que `B = {[n2,n1]}` y `K = {}`. O sea que el dominio de `Birthday` (o `B`) no es igual a `Known` (o `K`). Pero al mismo tiempo más arriba demostramos que `dom(Birthday,Known)` es un invariante. O sea que {*log*} devuelve una solución que no verifica el *otro* invariante. Cuando se da esta situación tenemos que agregar el otro invariante como hipótesis del lema de invarianza que está fallando:

```
neg(
  birthdayBookInv(K,B) &        % hipótesis añadida
  pfunInv(B) & addBirthday(K,B,N,C,K_,B_) implies pfunInv(B_)
).
```

¿Por qué falla 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
3. Falta algún otro invariante como hipótesis

Explicación del contraejemplo complejo

Cuando quisimos demostrar que `addBirthday` preserva `pfunInv` {*log*} nos devolvió un contraejemplo bastante complejo:

```
B = {[N,_N2]/_N1},
K_ = {N/K},
B_ = {[N,_N2],[N,C]/_N1}
Constraint: pfun(_N1), comppf({[N,N]},_N1,{}), N nin K,
            [N,C] nin _N1, C neq _N2, [N,_N2] nin _N1,
```

Acá vamos a explicar cómo entender esta solución. `_N1` y `_N2` son variables nuevas. Las variables nuevas deben interpretarse como variables cuantificadas existencialmente. O sea que la solución se debe entender en términos de "existen `_N1` y `_N2` tales que. . . ". El predicado `comppf({[N,N]},_N1,{})` dice que en `_N1` no hay ningún par ordenado cuya primera componente sea `N`. A su vez `_N1` es el resto de `B_`. O sea que en `B_` todos los pares ordenados con primera componente `N` son `[N,_N2]` y `[N,C]`. Al pedir que `C neq _N2` {*log*} está diciendo que `[N,_N2]` y `[N,C]` son dos pares ordenados distintos. Esto significa que `B_` no es una función porque tiene dos pares ordenados distintos con la misma primera componente.
En definitiva {*log*} encuentra valores para las variables de estado anterior y de entrada que producen un estado final donde `B_` no es una función, lo cual es un contraejemplo de la propiedad que queremos demostrar.

Además de demostrar que cada operación del sistema preserva el invariante es necesario demostrar que el estado inicial lo satisface (porque de lo contrario el sistema comenzaría a ejecutar en un estado que viola el invariante y en consecuencia las operaciones no tendrían nada que preservar). En general esta demostración es mucho más simple. Dado que lo único que tenemos que hacer es ver que el estado inicial satisface cada invariante, no necesitamos calcular la negación de ninguna parte de la fórmula. En las pruebas de satisfacibilidad del invariante esperamos que {*log*} responda con una solución (si responde `no` significa que el estado inicial no satisface el invariante). Comenzamos demostrando que el estado inicial satisface dom *birthday* = *known*:

```
birthdayBookInit(K,B) & dom(B,K).
```

Finalmente demostramos que en el estado inicial *birthday* es una función parcial:

```
birthdayBookInit(K,B) & pfun(B).
```

4.2. Pruebas que involucran aritmética entera

Hasta el momento en ninguna de las pruebas que hicimos tuvimos que usar aritmética (entera). Cuando tenemos que usar aritmética, tenemos que activar CLP(Q) antes de hacer las pruebas porque de lo contrario la respuesta de {*log*} no es fiable. Esto se puede ver con este simple ejemplo.

```
{log}=> int_solver(clpfd).
```

```
{log}=> Y < X + 1 & X + 1 < Y.
***WARNING***: non-finite domain
true
Constraint: integer(X), integer(Y)
```

Como pueden ver, aunque `Y < X + 1 & X + 1 < Y` es claramente insatisfasible en $\mathbb{Z}$, {*log*} es incapaz de darse cuenta porque el *solver* activo para aritmética entera es CLP(FD). Por el contrario, lo siguiente funciona correctamente:

```
{log}=> int_solver(clpq).

{log}=> Y < X + 1 & X + 1 < Y.

no
```

> CLP(Q) está activo por defecto en {*log*}.

4.3. Chequeo de tipos y demostraciones

Como ocurre con las simulaciones, las demostraciones se pueden hacer con el control de tipos activado o desactivado. Todo lo hecho más arriba asume que el control de tipos está desactivado (comando `notype_check`). Al igual que con las simulaciones, activar el control de tipos torna un poco más engorrosas las demostraciones (porque hay que dar un tipo para cada variable) pero a la vez las torna más seguras (porque el control de tipos detecta errores que el núcleo deductivo de {*log*} no puede detectar).

Como al momento de verificar el sistema las demostraciones son más críticas que las simulaciones, se recomienda activar el control de tipos al ejecutar los lemas de invarianza.

Mostraremos, entonces, cómo ejecutar demostraciones cuando el control de tipos fue activado (comando `type_check`). Si ejecutamos la prueba sin declarar los tipos de las variables:

```
neg(
  birthdayBookInv(K,B) & addBirthday(K,B,N,C,K_,B_)
  implies birthdayBookInv(K_,B_)
).
```

obtenemos un error de tipos:

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

Entonces, debemos dar el tipo para cada variable usando el predicado `dec`:

```
dec([B,B_],bb) & dec([K,K_],kn) & dec(N,name) & dec(C,date) &
neg(
  birthdayBookInv(K,B) & addBirthday(K,B,N,C,K_,B_)
  implies birthdayBookInv(K_,B_)
).
```

en cuyo caso la respuesta es no. Claramente, si la declaración de tipos es errónea:

```
dec([B,B_],bb) & dec([K,K_],kn) & dec(N,date) & dec(C,name) &
neg(
  birthdayBookInv(K,B) & addBirthday(K,B,N,C,K_,B_)
  implies birthdayBookInv(K_,B_)
).
```

obtenemos un error de tipos:

```
***ERROR***: type error:
  in addBirthday(K,B,N,C,K_,B_) arguments have the wrong type:
    N is date but should be name
    C is name but should be date
```

4.4. 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. Leer la sección 12 del manual del usuario de {*log*} para una descripción 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 con el mismo nombre de archivo. Para la especificación de la agenda de cumpleaños es así:

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

El VCG controla que se cumplan ciertas restricciones en la descripción de la máquina de estados tales como que haya una cláusula variables, las cláusulas invariant se ubiquen antes de las cláusulas operation, etc. Si alguna de estas restricciones no se cumple se imprime el error correspondiente. El VCG también controla la presencia de *variables unitarias* o *singleton variables*. Una variable unitaria es una variable que se usa solo una vez en la cláusula. 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.pl. 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.pl').
```

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
```

```
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

Total VCs: 11 (discharged: 10, failed: 1, timeout: 0)
Execution time (discharged): 0.0055484771728515625 s
```

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. En el caso de la agenda de cumpleaños el problema con `addBirthday_pi_pfunInv` es el que analizamos más arriba: falta una hipótesis para poder demostrar que `addBirthday` preserva `pfunInv`. Para corregir este error hacemos lo siguiente:

1. Editar el archivo `bb-vc.pl`.
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. Consultar el archivo `bb-vc.pl` nuevamente.
5. Ejecutar el comando `check_vcs_bb` nuevamente.

Si el error no se debiera a la falta de una hipótesis y en consecuencia tuviésemos que modificar el archivo con la especificación, probablemente deberíamos volver a ejecutar el comando `vcg`[21]. 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.

[21] Esto hay que hacerlo cuando modificamos el nombre de alguna cláusula o su aridad; o cuando eliminamos o agregamos cláusulas.

Finalmente, para facilitar la búsqueda de hipótesis faltantes se puede recurrir al comando `findh` que se explica en la sección 12.4 del manual del usuario. Este comando es útil *solo* cuando el lema de invarianza no se puede probar porque faltan hipótesis. Si la causa que impide probar el lema es de otra naturaleza (ver más arriba el cuadro "¿Por qué falla un lema de invarianza?"), `findh` no les va a resultar útil.

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. A decision procedure for a theory of finite sets with finite integer intervals. *ACM Trans. Comput. Logic*, sep 2023.

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

[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).

dec_p_type(n_birthdayBookInv(kn,bb)).
n_birthdayBookInv(Known,Birthday) :- neg(dom(Birthday,Known)).

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

dec_p_type(n_pfunInv(bb)).
n_pfunInv(Birthday) :- neg(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,kn,bb)).
findBirthday(Known,Birthday,Name,Date,Known,Birthday) :-
  findBirthdayOk(Known,Birthday,Name,Date)
  or
```

```
  notAFriend(Known,Name).

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