Universidad Complutense de Madrid
Facultad de Ciencias Matemáticas
Grado en Ciencias Matemáticas

Trabajo Fin de Grado

Test de unidad en Maude

José Ignacio Pérez Valverde

Tutor
Adrián Riesco

Marzo 2017

Abstract

A lo largo de este proyecto de fin de grado crearemos distintos módulos de geometría euclídea con regla y compás en Maude acompañándolos de sus correspondientes test de unidad, los cuales se usarán para crear un método de aprendizaje para el lenguaje. Para ello comenzaremos dando unas nociones básicas sobre Maude, el lenguaje declarativo que vamos a utilizar, explicando entonces sus distintos modelos de programación, véanse, funcionales y de sistema. A continuación presentaremos mUnit, el módulo auxiliar encargado de realizar los test de unidad, y explicaremos el uso y funcionamiento de todos los comandos que nos proporciona. El proyecto se encuentra dividido en tres partes. La primera de ellas se corresponde con la creación de los módulos referentes a la geometría, los cuales nos permitirán ver los dos tipos de módulos que podemos implementar Maude, incluido uno interactivo que nos permitirá emular la geometría con regla y compás. La segunda parte se encuentra formada por los test de unidad definidos para cada módulo, y dentro de ellos para cada función, de los definidos en la primera parte. Finalmente, la tercera parte constará de unos módulos de aprendizaje, los cuales se apoyarán en todo lo anteriormente creado para facilitar a los que estén interesados en el lenguaje a comenzar con él.

In this project we will create different modules of Euclidean geometry with ruler and compass in Maude accompanied by unit test, whom will be used to create a learning method for the language. For this purpose we start with some basics notions about Maude, the declarative language that we will work with, explaining then its differents modules, functionals and systems. After that, we will present mUnit, the auxiliar module that will make the unit test, and we will explain the used and the operation of all the mUnit commands. The project is divided in three parts. The first one refers to the creation of the geometry's modules we want to create in Maude, they will system and functional modules, one interactive that emulates the ruler and compass geometry will be included. The second part is formed by the unit test that will be defined for every module and ever function of the first part. Finally, the third part will consist of the learning modules, whom will use all that we have create before to ease the learning of anybody interested in Maude to start with it.

Índice general

Capítulo 1

Introducción

En esta memoria se presenta el lenguaje declarativo Maude a lo largo de diversos ejemplos y sus correspondientes test de unidad. Valiéndonos de estos últimos aprovecharemos también para crear distintos módulos de aprendizaje para aquellos que deseen iniciarse en este lenguaje.

A lo largo del Grado en Ciencias Matemáticas, más concretamente en la rama de Matemática Computacional, hemos estudiado diversos lenguajes de tipos muy diferentes, comenzando por Python y Pascal y llegando a Haskell y Prolog, que poco tienen que ver con los primeros. Los primeros son lenguajes imperativos mientras los últimos son declarativos. Así pues, Maude, un lenguaje para mí desconocido resultaba una propuesta interesante.

Además para dotar de mayor consistencia al proyecto, no solo se implementará el lenguaje mediante implementación de ejemplos, también se desarrollarán test de unidad que permitirán evaluar los conocimientos de futuros estudiantes. Estos son los otros desconocidos del proyecto que aquí se expone, pues aunque de forma abstracta hayamos podido trabajar con ellos, en el sentido de probar manualmente las distintas funciones, la verdad es que el mayor contacto que se tiene con test de unidad propiamente dichos es en asignaturas en que nuestros programas deben pasarlos, pero nunca llegamos a implementarlos.

Ahora sí, con estos dos objetivos en mente, se decidió crear una serie de módulos, que nos mostraran claramente qué es Maude, cómo funciona, y que al mismo tiempo nos permitiesen crear una buena variedad de pruebas. Aquí es importante decir que Maude es un lenguaje algebraico, y como tal, gran parte de los ejemplos que se pueden encontrar consisten en distintos modelos de este tipo, como anillos o cuerpos. Debido a esto se eligió algo distinto, que fuese claro al verse desde fuera, que pudiese tener una aplicación futura, y que además nos permitiese crear test de unidad lo suficientemente variados y que no resultara demasiado abstracto para implementar un método de aprendizaje. Por estos motivos se decidió implementar la geometría clásica en dos dimensiones, y posteriormente un sistema de entrada/salida que emulase las construcciones geométricas con regla y compás.

Así pués, en los siguientes capítulos se procederá a explicar Maude y la creación de los módulos dedicados que crearemos así como los ya mencionados test de unidad y módulos de aprendizaje. En el capítulo 2 se explicará qué es Maude con sus dos módulos principales de programación y numerosos ejemplos, a continuación presentaremos los test de unidad explicando las herramientas de que disponemos y de nuevo dando ejemplos. Concluyendo el capítulo explicaremos en qué consiste el método de aprendizaje y su relación con otros ya existentes para otros lenguajes. En el capítulo 3 programaremos los módulos de geometría en dos dimensiones comenzando por puntos y rectas, y concluyendo estos con la definición de las circunferencias y un último módulo llamado `GEO2D` que se encargará de juntar todas las figuras anteriores. Con la idea de la entrada y la salida, emulando la regla y el compás, creamos el capítulo 4, pero antes creamos en el mismo capítulo los diccionarios de figuras, que serán la estructura de datos que utilizaremos en el séptimo para almacenar la información. Finalmente, en el capítulo 5,

se darán unas conclusiones finales y se expondrán las distintas maneras de continuar con el proyecto, ya sea extendiendo la geometría, creando un constructor automático, o simplemente demostrando la corrección y la completitud de las funciones implementadas.

Todos los módulos programados para el proyecto podrán encontrarse en el siguiente repositorio online (incluir dirección), donde hay 13 módulos para Maude y 10 test de unidad dando un total de 1306 lineas de código, sin contar los 23 módulos de aprendizaje.

Capítulo 2

Preliminares

A lo largo de este capítulo se explicará qué son Maude, los test de unidad y los módulos de aprendizaje, aprovechando lo cual se pasará a dar ejemplos de los primeros módulos con sus casos de test. Asimismo, haremos también mención a las herramientas que utilizaremos a los largo del proyecto.

2.1. Maude

Maude (Clavel et al., 2007) es un lenguaje declarativo, véase, sus programas se construyen declarando condiciones, ecuaciones o transformaciones que definen el problema y nos dan una solución, la cual no será construida siguiendo un conjunto de pasos sino que será el intérprete el encargado de deducirla en base a las funciones definidas. Maude presenta dos tipos distintos de módulos, que nos permiten modelizar distintos tipos de problemas.

El primero de estos es el módulo funcional. En él daremos una serie de tipos, subtipos y relaciones entre ellos, las cuales estarán dadas en forma de ecuaciones que, aunque denotan igualdades, desde el punto de vista del cómputo se ejecutan de izquierda a derecha. Además de esto nos encontraremos tanto en los operadores como en las relaciones o los constructores distintos axiomas, como la conmutatividad o la asociatividad. De esta manera, cuando Maude interpreta estos módulos lo que hace es utilizar las distintas ecuaciones para deducir un resultado encajando los términos módulo el conjunto de axiomas.

Por otro lado nos encontramos con los módulos de sistema. Estos funcionan de manera similar a los funcionales con la salvedad de que, además de tener relaciones dadas por ecuaciones, permite definir transiciones. Estas estarán dadas por diferentes reglas que, de manera similiar a las ecuaciones de los módulos funcionales, se ejecutarán de izquierda a derecha, pero en este caso no denotan igualdades sino cambios de estado.

Para ilustrar estos conceptos se procederá a la creación de dos ejemplos.

2.1.1. Ejemplo de módulo funcional

Como ejemplo de módulo funcional implementaremos los naturales de Peano con diferentes operaciones. En Maude los módulos funcionales se crean siguiendo el siguiente esquema: `fmod` ⟨Nombre del Módulo⟩ `is` ⟨Definiciones y Declaraciones⟩ `endfm`. Así, el módulo `PEANO-NAT-EXTRA` comienza como sigue:

```
fmod PEANO-NAT-EXTRA is
```

Es posible definir tipos de datos mediante `sort`, y `sorts` en el caso en que sean varios. Por ejemplo, en el caso de los naturales (`Nat`), los naturales distintos de cero (`NoZeroNat`) y el cero (`ZeroNat`) se definen como:

```
sorts Nat NoZeroNat ZeroNat .
```

Dentro de los tipos se les puede dotar de orden con la palabra reservada `subsort`, por ejemplo:

```
subsort NoZeroNat < Nat .
subsort ZeroNat < Nat .
```

En este caso solo definiremos los `ZeroNat` y `NoZeroNat`, sin embargo en las funciones el dominio serán siempre los `Nat`. Retomando la definición de estos, antes de utilizarlos suele ser necesario darles un constructor para poder trabajar con ellos. Para ello utilizaremos la palabra reservada `op`, los tipos de los argumentos y del resultado, y por último indicaremos que es un constructor con el atributo `ctor`.

```
op 0 : -> ZeroNat [ctor] .
op s : Nat -> NoZeroNat [ctor iter] .
```

Como se puede observar, a los constructores se les puede dotar de axiomas como `iter`, que le permite acortar los términos construidos con ese operador, por ejemplo, `s(s(0))` sería lo mismo que `s^2(0)`, otros axiomas importantes serían `comm`, `assoc` que le dotarían de las propiedades de commutatividad y asociatividad respectivamente. Continuando con la creación del módulo, antes de definir ninguna función será necesario hacer una declaración de variables en base a sus tipos mediante el comando `var`, o `vars` terminando con el tipo correspondiente.[1]

```
vars M N R : Nat .
```

La creación de las funciones es igual a la de los constructores, pero en este caso no tendrán el axioma `ctor` y contarán con ecuaciones. Por otro lado hay que dotar a las funciones de forma, esto se puede hacer simplemente indicando un nombre o mediante una construcción del tipo `_+_` donde `_` indica la posición de los valores de entrada, quedando entonces como sigue:

```
op add : Nat Nat -> Nat [comm assoc] .     op _+_ : Nat Nat -> Nat [comm assoc] .
```

Una vez definida la función habrá que dotarla de comportamiento mediante ecuaciones, las cuales como ya se ha comentado antes, se ejecutarán de izquierda a derecha. Estas comenzarán por `eq`, o `ceq` para ecuaciones condicionales, e indicarán mediante una igualdad el resultado de la operación. A la hora de dar las ecuaciones dependemos de la definición de la operación, siendo las ecuaciones del primer caso para `add` izquierda, y las del segundo para `_+_`:

```
eq add(0 , N) = N .                        eq 0 + N = N .
eq add(s(M) , N) = s(add(M , N)) .         eq s(M) + N = s(M + N) .
```

Como se indicó anteriormente, es posible declarar variables al vuelo como sigue:

```
eq 0 + N:Nat = N:Nat .
eq s(M:Nat) + s(N:Nat) = s(M:Nat + N:Nat)
```

El método utilizado para dar las variables dependerá de las veces que se vaya a usar esta a lo largo del módulo, pero por comodidad se seguirá utilizando la declaración de variables inicial. Retomando las funciones, esta no es la única forma que tenemos de darlas, sino que en los casos de condicionales tendremos que usar el comando `ceq` así como dar con `if` las condiciones necesarias, las cuales se unen mediante el operador `_/\_`. Estas pueden ser expresiones Booleanas, comprobaciones de tipo y condiciones de encaje de patrones. A continuación mostraremos condiciones Booleanas, más información está disponible en (Clavel et al., 2007)

[1] También existe la posibilidad de declarar las variables al vuelo en las propias ecuaciones de la función, como se vera más adelante.

```
op _*_ : Nat Nat -> Nat [comm] .
ceq M * N = 0
if M == 0.
ceq s(M) * N = (M * N) + N
if not (s(M) == 0) .
```

De la misma manera creamos la igualdad, `_eq_`, y la función menor que, `_<_`.

```
op _eq_ : Nat Nat -> Nat [comm] .
eq 0 eq s(M) = false .
eq N eq N = true .
eq M eq N = false [owise] .
```

```
op _<_ : Nat Nat -> Bool .
eq 0 < s(M) = true .
eq s(M) < s(N) = M < N .
eq M < N = false [owise] .
```

Se puede ver en ambas funciones la aparición del atributo `[owise]` en una ecuación de cada una. La función de este es indicar que, en caso de no encajar los atributos de entrada con ninguna de las ecuaciones anteriores, se deberá ejecutar esta.

Una vez hayamos creado e incluido todas las funciones será necesario indicar la terminación del módulo mediante:

```
endfm
```

Veamos ahora un ejemplo de ejecución del módulo funcional que acabamos de crear. Para comenzar deberemos cargar el módulo desde la consola de Maude.

```
Maude> load PEANO-NAT-EXTRA.maude
```

El intérprete de Maude no ejecuta los programas como tal, sino que lee lo que hayamos escrito y lo reduce lo máximo posible utilizando las ecuaciones implementadas. Deberemos escribir `red` ⟨Expresión que queremos reducir⟩ `.`, teniendo cuidado con el punto. Veamos algunos casos sencillos:

```
Maude> red s(0) .
reduce in PEANO-NAT-EXTRA : s(0) .
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
result NoZeroNat: s(0)
```

Como se puede observar Maude indica el tipo mínimo del término resultado (`NoZeroNat`), que en este caso es el mismo que el inicial porque no se le pueden aplicar ecuaciones. Este también nos indica el tiempo de ejecución.

Aunque la función que hemos ejecutado resulta sencilla podemos ver que todo funciona correctamente, veamos otro caso más, este sí con mayor complejidad al tener que aplicar al menos dos ecuaciones:

```
Maude> red (s(0) + s(s(0))) * s(s(0)) .
reduce in PEANO-NAT-EXTRA : (s(0) + s^2(0)) * s^2(0) .
rewrites: 15 in 0ms cpu (0ms real) (~ rewrites/second)
result NoZeroNat: s^6(0)
```

En este caso vemos que lo primero que ha hecho Maude ha sido utilizar el axioma `iter` para reducir las expresiones y después reducir la expresión, lo cual ha realizado con 15 reescrituras.

2.1.2. Ejemplo de módulo de sistema

El módulo que crearemos de ejemplo para los módulos de sistema será una representación del problema de las vasijas (Clavel et al., 2007). Este problema consiste en, dadas n vasijas con distintas capacidades, las cuales podermos llenar del todo, vaciar cuando queramos y verter de una a otra hasta vaciar la primera o llenar la segunda, conseguir que en una de ellas nos quede una cantidad de líquido concreta. El caso más común es aquel en que tenemos 3 recipientes con una capacidad de 3, 5 y 8 litros respectivamente, y buscamos conseguir que en uno de ellos queden 4 litros de agua.

En Maude los módulos de sistema se crean siguiendo el siguiente esquema: `mod` ⟨Nombre del Módulo⟩ `is` ⟨Definiciones, Declaraciones y Reglas⟩ `endm`. Empezaremos definiendo el módulo `VASIJAS`:

```
mod VASIJAS is
```

Los comandos básicos, como importar, dar los tipos, constructores y variables mantienen la misma sintaxis que en los módulos funcionales. Como necesitamos un conjunto númerico para la creación del tipo `Vasija` comenzaremos importando los números naturales. Para importar un módulo tenemos tres opciones, `protecting`, `extending` y `including`, sin embargo en general utilizaremos `protecting`, de la forma `pr`, ya que nos importa los módulos pero no nos permite modificarlos, evitando así confusión y problemas de interacción con otros. Así pues, importamos el módulo predefinido `NAT`.

```
protecting NAT .
sort Vasija CjVasija .
subsort Vasija < CjVasija .

op vasija : Nat Nat -> Vasija [ctor] .
*** El primer numero se corresponde con la cantidad de liquido que contiene
    la vasija y el segundo con su capacidad maxima.
op _,_ : Vasija Vasija -> CjVasija [ctor comm assoc] .

vars C1 C2 N1 N2 : Nat .
```

Como se puede ver las vasijas están definidas como un par, en el que el primer elemento se corresponde con la cantidad de líquido que contiene y el segundo con la capacidad total de esta. Aparte del tipo `Vasija` creamos tambien el conjunto de estas, `CjVasijas`, en el que almacemaremos varias vasijas a la vez.

Una vez completados los constructores y la declaración de variables si se desea, procedemos a construir el módulo en sí. Para ello, aparte de las reglas que veremos más adelante, podemos crear funciones mediante ecuaciones al igual que hacíamos en los módulos funcionales. En nuestro caso la única que definiremos será sobre la que queremos que funcionen las reglas, véase, las tres vasijas antes comentadas vacías.

```
op inicial : -> CjVasija .
*** Da un conjunto inicial de vasijas vacias con distintas capacidades.
eq inicial = vasija(0, 3) , vasija(0, 5) , vasija(0, 8) .
```

Hecho esto podemos ponernos con las reglas en sí. Estas deben comenzar por `rl`, y de manera opcional a continuación se las puede identificar mediante `[_]` donde `_` es el nombre que damos a la regla. Después esta se construye como una transición de izquierda a derecha dada por `=>`, por ejemplo la regla `vacia`, que se encarga de vaciar un vasija se daría como sigue:

```
rl[vacia] : vasija(N1, C1) => vasija(0, C1) .
***Esta regla nos vacia la vasija.
```

De manera análoga a la anterior creamos la regla `llena`, que nos llena la vasija hasta su capacidad máxima.

```
rl[llena] : vasija(N1, C1) => vasija(C1, C1) .
*** Esta regla nos llena la vasija al maximo.
```

Por supuesto, igual que ocurría con las ecuaciones, podemos construir reglas condicionales. En este caso comenzarían por `crl` y se construirían igual que las anteriores, indicando eso sí mediante un `if` la condición que deben cumplir.

En nuestro caso crearemos las funciones que nos transfieren el líquido de una vasija a la siguiente.

```
crl[transferir1] : vasija(N1, C1) , vasija(N2, C2)
                 => vasija(0, C1) , vasija((N1 + N2), C2)
if (N1 + N2) <= C2 .
*** Nos transfiere el contenido de una vasija a otra si la suma de
    los contenidos no se pasa de la capacidad de la segunda.

crl[transferir2] : vasija(N1, C1) , vasija(N2, C2)
                 => vasija(sd((N1 + N2) , C2), C1) , vasija(C2, C2)
if (N1 + N2) > C2 .
*** Nos transfiere el contenido de una vasija a otra si la suma de
    los contenidos se pasa de la capacidad de la segunda.
```

Aunque las dos reglas se refieren a la misma acción cubren casos distintos, siendo la primera aquella en que el contenido de las dos vasijas cabe en la segunda, a la que se quiere transferir, y la segunda aquella en la que no, quedando la primera entonces con el líquido sobrante. El módulo finalmente se cierra con `endm`.

```
endm
```

Veamos ahora un ejemplo de ejecución del módulo de sistema que acabamos de crear. Por supuesto debemos comenzar cargando el módulo VASIJAS :

```
Maude> load VASIJAS.maude
```

Además del comando `red`, que ya vimos en los módulos funcionales, los de sistema tienen más opciones disponibles. El comando básico es `rewrite`, abreviado `rew`, que aplica las ecuaciones y las reglas del módulo al término dado como argumento. Sin embargo, este ejemplo no termina (es posible vaciar todo el tiempo una vasija vacía, por ejemplo), así que tendremos que indicar manualmente el número de reglas que queremos utilizar en total, por ejemplo 10:

```
Maude> rew [10] inicial .
rewrite [10] in VASIJAS : inicial .
rewrites: 139 in 0ms cpu (5ms real) (~ rewrites/second)
result [CjVasija]: vasija(0, 3),vasija(0, 5),vasija(0, 8)
```

Al parecer no ha ocurrido nada, probablemente porque Maude haya utilizado diez veces la regla que vacía una vasija, así que hay que obligarle a utilizar más reglas. Para ello utilizaremos el commando `frew`, fair rewrite o reescritura justa.

```
Maude> frew [10] inicial .
frewrite in VASIJAS : inicial .
rewrites: 22 in 0ms cpu (201ms real) (~ rewrites/second)
result (sort not calculated): vasija(0, 8),vasija(3, 3),vasija(5, 5)
```

Podemos ver que en este caso se han llenado las dos ultimas vasijas, pero esto tampoco nos aporta mucha información. Para obtener información relevante necesitamos buscar una solución concreta y que Maude nos la construya, pudiendo consultar después los pasos que ha dado. Para esto utilizaremos el comando `search`

```
Maude> search [1] inicial =>* vasija(4, N:Nat) , B:CjVasija .
```

Antes de mostrar la ejecución explicaré lo escrito. `[1]` indica que solo buscamos una solución, `inicial =>*` nos indica que utilizamos la función `inicial` partiendo de ella, y que el estado final lo queremos alcanzar en cualquier número de pasos (las otras opciones son `=>+`, que requiere 1 o más pasos, `y =>!`, que solo devuelve estados finales, es decir, expresiones a las que no se les pueden aplicar ecuaciones ni reglas), por último `N:Nat` y `B:CjVasija` son variables que nos indican el tipo de los datos que deben ocupar esa posición, pero que pueden tomar cualquier valor. Veamos ahora la ejecución.

```
Maude> search [1] inicial =>* vasija(4, N:Nat) , B:CjVasija .
search in VASIJAS : inicial =>* B:CjVasija,vasija(4, N:Nat) .

Solution 1 (state 75)
states: 76  rewrites: 2142 in 20ms cpu (48ms real) (107100 rewrites/second)
B:CjVasija --> vasija(3, 3),vasija(3, 8)
N:Nat --> 5
```

Como se puede ver la solución encontrada es aquella en la que la vasija con cuatro litros es aquella que tenía cinco de capacidad, y las otras dos quedan con tres litros cada una. Esta solución ha sido alcanzada en el estado 75, y se puede consultar con el comando `show path 75`, que nos indica cuándo y dónde se ejecutaron qué reglas:

```
Maude> show path 75 .
state 0, [CjVasija]: vasija(0, 3),vasija(0, 5),vasija(0, 8)
===[ rl vasija(N1, C1) => vasija(C1, C1) [label llena] . ]===>
state 2, [CjVasija]: vasija(0, 3),vasija(0, 8),vasija(5, 5)
===[ crl vasija(N1, C1),vasija(N2, C2) => vasija(C2, C2),vasija(sd(C2, N1 +
    N2), C1) if N1 + N2 > C2 = true [label transferir2] . ]===>
state 9, [CjVasija]: vasija(0, 8),vasija(2, 5),vasija(3, 3)
===[ crl vasija(N1, C1),vasija(N2, C2) => vasija(0, C1),vasija(N1 + N2, C2) if
    N1 + N2 <= C2 = true [label transferir1] . ]===>
state 20, [CjVasija]: vasija(0, 3),vasija(2, 5),vasija(3, 8)
===[ crl vasija(N1, C1),vasija(N2, C2) => vasija(0, C1),vasija(N1 + N2, C2) if
    N1 + N2 <= C2 = true [label transferir1] . ]===>
state 37, [CjVasija]: vasija(0, 5),vasija(2, 3),vasija(3, 8)
===[ rl vasija(N1, C1) => vasija(C1, C1) [label llena] . ]===>
state 55, [CjVasija]: vasija(2, 3),vasija(3, 8),vasija(5, 5)
===[ crl vasija(N1, C1),vasija(N2, C2) => vasija(C2, C2),vasija(sd(C2, N1 +
    N2), C1) if N1 + N2 > C2 = true [label transferir2] . ]===>
state 75, [CjVasija]: vasija(3, 3),vasija(3, 8),vasija(4, 5)
```

2.1.3. Core Maude y Full Maude

Core Maude es el intérprete del lenguaje implementado en C++ y que nos proporciona todas las funcionalidades básicas de Maude, todos los módulos desarrollados anteriormente en los ejemplos están realizados para poder ejecutarse en Core Maude directamente. Full Maude por otro lado es una extensión de Maude programada en el propio lenguaje que se encarga de proporcionarnos más herramientas para el desarrollo de módulos y que deberá ser cargada, ya sea en el fichero que se vaya a utlizar o desde la propia consola de Maude como si fuese un módulo más. Full Maude nos proporciona nuevas opciones muy interesantes a la hora de crear nuestros programas dándonos por ejemplo la posibilidad de utilizar la entrada/salida de datos, con la que trabajaremos mucho, o la creación de los módulos orientados a objetos, que no trataremos en este proyecto.

2.2. Los test de unidad y mUnit

Un test de unidad (Osherove, 2014) es un fragmento de código que llama a una función, o unidad de trabajo, y comprueba el resultado final de los programas o funciones que contiene. En caso de que uno de los casos resulte falso diremos que el test ha fallado.

Un test de unidad bien diseñado debe cumplir las siguientes condiciones (Osherove, 2014):

- Debe estar automatizado.
- Debe ser fácil de implementar.
- No debe quedar obsoleto.
- Ejecutarlo debe ser tan sencillo como pulsar un botón.
- Debe tener una ejecución rápida.
- Debe ser consistente con sus resultados.
- Debe tener control total sobre el módulo donde se ejecuta el test, véase, ejecutar todos los casos posibles en todas las funciones con que se encuentre.
- Debe ser independiente de la ejecución de otros test.[2]
- En caso de que falle, debe ser sencillo encontrar donde se originó el error.

.

Para trabajar con los test utilizaremos una herramienta ya creada llamada mUnit (incluir referencia web), que nos proporciona distintas herramientas para verificar el funcionamiento de las funciones. Estas son: `assertEqual`, `assertDifferent`, `assertTrue`, `assertFalse`, `assertTrueMsg`,
`assertReachable`, `assertReachableBnd` y `assertSort`, y realizarán las pruebas lógicas según sus nombres, estas se explicarán más adelante con ejemplos. Veamos a continuación ejemplos de los test de unidad en que utilizaremos dichas funciones:

Los test deben comenzar indicando el módulo sobre el que vamos a trabajar, en nuestro caso `PEANO-NAT-EXTRA`.

```
(munit PEANO-NAT-EXTRA is
```

[2]Esto no ocurre estrictamente en nuestro caso, porque tenemos E/S y es necesarío volver a los valores iniciales despues de cada ejecución. Esto se explicará con mayor claridad en el capítulo correspondiente.

El primer caso de test con que usaremos será `assertEqual(_,_)`, que evaluará las dos expresiones que recibe como datos y devolverá `true` en el caso de que sean iguales, o `false` en el caso de que no se evalúen a lo mismo. Por ejemplo:

```
assertEqual(s(0) + s(0), s(s(0)))
```

De forma complementaria nos encontramos con `assertDifferent(_,_)`, que evaluará a `true` si las expresiones evaluan a cosas distintas, y a `false` en otro caso. Por ejemplo:

```
assertDifferent(0, s(0))
```

Sin embargo no todas las operaciones se evalúan a una expresión, sino que es muy común que lo hagan a un Booleano. Debido a esto utilizaremos también otras dos funciones, `assertTrue(_)` y `assertFalse(_)`. Devolviéndonos la primera `true` en caso de que evalúe a `true` o `false` en otro caso. La segunda función se comporta al contrario, devolviéndonos `true` si la expresión evalua a `false`, o `false` en caso contrario. Existe también una función que es una variante de `assertTrue(_)`, `assertTrueMsg(_,_)`, que, como su nombre indica, nos permite añadir un comentario a la función para que se vea en la ejecución. Por ejemplo:

```
assertTrue(0 < s(0))
assertFalse(s(0) < 0)
assertTrueMsg(0 > 0 , "Deberia fallar")
```

También puede darse el caso de que la información necesaria sea el tipo de una expresión en particular, para lo cual utilizaremos `assertSort(_,_)` que recibe como parámetros una expresión y un tipo, y devuelve `true` en el caso de sea el tipo de la expresión y `false` en el otro. Por ejemplo:

```
assertSort(s(0), Nat)
```

Finalmente el módulo de test se cierran con `endu`.

```
endu)
```

Por otro lado los módulos de sistema nos permiten usar, además de todas las anteriores, unas dos últimas funciones, que trabajan únicamente con reglas: `assertReachable` y `assertReachableBnd`. La primera de ellas, `assertReachable(_,_)`, recibe dos expresiones, y nos indica si a partir de la primera se puede alcanzar la segunda con cualquier número de pasos. La segunda, `assertReachableBnd(_,_,_)`, nos indica lo mismo pero poniendo además un tope a los pasos, es decir, si desde primer estado se puede alcanzar el segundo en como mucho n pasos. Por ejemplo:

```
(munit VASIJAS is
   assertReachable(vasija(0, 3), vasija(3, 3))
   assertReachableBnd(vasija(0, 3), vasija(3, 3), 1)
endu)
```

Construido el test solo queda evaluarlo en Maude, para lo cual basta con cargarlo en la consola, lo cual en el caso de `PEANO-NAT-EXTRA` nos daría lo siguiente:

```
Maude> load testNat.maude

    Full Maude 2.7 March 10th 2015

MUnit: a unit testing framework for Maude.
Version 1.0(November 23rd, 2016)


6 test cases were executed.
1 failures.

assertEqual(s(0)+ s(0),s(s(0))) passed.
assertDifferent(0,s(0)) passed.
assertTrue(0 < s(0)) passed.
assertFalse(s(0)< 0) passed.
assertTrue(0 > 0) failed.
 --->  "Deberia fallar"
assertSort(s(0),Nat) passed.
```

2.3. Método de aprendizaje y MaudeKoan

Además de la implementación de los distintos módulos y sus test de unidad, se construirán de forma paralela unos módulos de aprendizaje, los cuales se encargarán de mostrar a través de distintos casos cómo ir definiendo los distintos tipos, sub-tipos y operadores que se utilizan en Maude.

Estos módulos de aprendizaje se construirán una vez finalizados el módulo y el test de unidad correspondiente. Serán similares a los módulos completos, con la salvedad de que presentarán huecos, que deberán rellenarse. Sin embargo una vez las funciones avancen, nos encontraremos con algunas que llevarán detrás unos cálculos que, aún siendo relativamente elementales, no se podrán dejar con huecos debido a su complejidad, así pues, en estos casos los huecos aparecerán en los propios test de unidad donde se deberá indicar que se ha comprendido correctamente el funcionamiento de las distintas funciones.

Sin embargo esta idea no es nueva, sino que se inspira claramente en el proyecto ScalaKoans, que nos ayuda a aprender Scala con un método muy similar, aunque no igual, y que podemos ver para otros muchos lenguajes como puede ser Python. ScalaKoans pertenece al proyecto Koans, cuya filosofia es la de facilitar el acceso a los lenguajes de programación mediante un entorno interactivo que permita a aquellos que quieran familiarizarse con los lenguajes con relativa facilidad. Las diferencias entre nuestro proyecto y ScalaKoans son las siguientes:

- Los casos no serań completados desde un programa externo, sino que deberemos modificar los módulos `.maude` manualmente.
- De manera análoga, no utilizaremos un programa externo para comprobar la vericidad de nuestros programas, sino que se tendrán que ejecutar los test de unidad desde la propia consola de Maude hasta conseguir que todas las funciones los pasen.

A continuación daremos un ejemplo de estos módulos incompletos, que no se corresponderá con `PEANO-NAT-EXTRA` o `VASIJAS`, sino que implementaremos un módulo nuevo, las listas. Al estar trabajando de nuevo con un módulo funcional este vuelve a empezar con `fmod`, sin embargo como queremos utilizar los naturales debemos importarlos.

```
fmod Lista is
   pr PEANO-NAT-EXTRA .
```

Ahora ya sí definimos el módulo comenzando por los tipos, las variables, y finalmente dos constructores, `mt` y `__`, correspondiendose el primero con la lista vacía y el segundo con la concatenación de dos naturales.

```
   sort Lista .
   subsorts Nat < Lista .
   var M : Nat .
   var L : Lista .

   op mt : -> Lista [ctor] .
   op __ : Lista Lista -> Lista [ctor assoc comm id: mt] .

endfm
```

Una de las cosas que debe llamarnos la atención es `id: mt` que nos proporciona un elemento neutro para la operación, en este caso `mt`, que identificaremos, como ya hemos dicho, con la lista vacía.

Por supuesto el módulo tal cual queda muy pobre, y no sería demasiado complicado completarlo con nuevas funciones como aquellas que nos indiquen si una lista es vacía o su longitud. Estas serán las operaciones que se dejarán con huecos para que sean completadas, los cuales estarán complementadas con su test correspondiente.

```
op empty-list : Lista -> Bool .
*** Comprueba si una lista es la
*** lista vacia.
eq empty-list(mt) = true .
eq empty-list(L M) = *** .
```

```
op length-list : List -> *** .
*** Da la longitud de una lista.
eq length-list(mt) = 0 .
eq length-list(L M) = s(length-list(L)) .
```

Veamos entonces el test para listas:

```
(munit Lista is
   assertEqual(mt, mt mt)
   assertEqual(s(0), mt s(0))
   assertEqual(s(0) s(s(0)), s(s(0)) s(0))
   assertTrue(empty-list(mt))
   assertFalse(empty-list(s(0)))
   assertEqual(length-list(mt), 0)
   assertEqual(length-list(s(s(0))), s(0))
   assertDifferent(length-list(s(s(0))), s(s(0)))
endu)
```

Ahora sí podemos ver que `empty-list(s(0))` debe darnos `false`, de la misma manera que el tipo de salida de `length-list` debe ser naturalmente `Nat`.

Con esto concluimos las nociones preliminares de Maude para el resto del trabajo.

Capítulo 3

Puntos, rectas y circunferencias

A lo largo de este capítulo desarrollaremos tres módulos dedicados enteramente a la geometría en dos dimensiones. Comenzaremos por crear los puntos y las rectas con todas las funciones que se considerarán necesarias, así como los correspondientes test de unidad. Posteriormente crearemos el dedicado a las circunferencias, que nos proporcionará la última de las figuras geométricas que necesitamos. Finalmente crearemos otro módulo que se encargará de cargar los tres anteriores y aunar en una única función aquellas que nos proporcionan puntos de corte, facilitándonos así el proceso posteriormente en el módulo referido a la entrada y la salida de datos. Por último aclarar que los módulos de aprendizaje por el contrario no aparecerán aquí, sino que podrán encontrarse en el repositorio online.

3.1. POINT

Comencemos con la creación de los puntos. La definición de estos es sencilla, construyéndose como un par formado por dos números reales, igual que se haría en papel. Las funciones que hemos considerado necesarias por el momento son únicamente dos, la que nos calcula la distancia entre dos de ellos `distance`, y la que nos intercambia las coordenadas `trans-point`. Así pues el módulo funcional, quedaría como sigue:

```
fmod POINT is

   pr FLOAT .
   sorts Point .

   op p(__) : Float Float -> Point [ctor] .
   *** Define un punto como un par de numeros reales

   op distance : Point Point -> Float .
   *** Calcula la distancia entre dos puntos mediante la metrica usual.
   eq distance(p(z11 z12) , p(z21 z22)) = sqrt(((z21 - z11) ^ 2.0)
                                          + ((z22 - z12) ^ 2.0)) .

   op trans-point : Point -> Point .
   *** Intercambia las coordenadas de un punto
   eq trans-point(p(z11 z12)) = p(z12 z11) .

endfm
```

El módulo queda entonces completo aunque resulta muy simple, sin embargo, como será necesario importarlo para la creación de los otros dos, lo utilizaremos también para las funciones

auxiliares que vayamos necesitando en los módulos siguientes, como por ejemplo las ecuaciones de segundo grado. Para concluir esta sección daremos algunos casos del test de unidad correspondiente a estas funciones, ilustrando así su funcionamiento. Para evitar saturar el capítulo con estos test solo se incluirán cuando contengan información nueva o sean inportantes. Veamos ahora sí algunos casos:

```
(munit POINT is
assertEqual(distance(p(1.0 0.0), p(0.0 1.0)), sqrt(2.0))
assertEqual(distance(p(0.0 0.0), p(4.0 3.0)), sqrt(13.0))
assertEqual(distance(p(1.0 0.0), p(2.0 0.0)), 1.0)
assertEqual(distance(p(1.0 1.0), p(1.0 3.0)), 2.0)
assertDifferent(trans-point(p(1.0 0.0)), p(1.0 0.0))
assertEqual(trans-point(p(1.0 0.0)), p(0.0 1.0))
endu)
```

3.2. LINE

Dando por finalizada la creación del `fmod POINT`, comenzaremos con el de las rectas. Estas se implementarán de forma análoga a los puntos, pero como un par de estos en vez de números reales. Como estamos siguiendo un camino similar al de la geometría básica, implementaremos además el tipo vector, que se utilizará en varias operaciones de las rectas. Estos se construirán de nuevo como un par de reales, que serán el correspondiente a extremo menos origen de los puntos que las definen. Empezamos dando los constructores correspondientes:

```
fmod LINE is

   pr POINT .
   sorts Line Vector .

   op r(__) : Point Point -> Line [ctor] .
   *** Define una recta como un par de puntos, mas adelante la daremos ecuaciones.
   op v(__) : Float Float -> Vector [ctor] .
   *** Define un vector como un par de reales.
   *** Extremo menos origen.
```

Las primeras funciones que crearemos para los vectores servirán para, o bien obtener información de ellos, o bien construir otros. Estas se corresponden con `v-scalar-prod`, que nos devuelve el producto escalar de dos vectores, `v-norm`, que nos da la norma de un vector, `perpendicular-vector`, que nos proporciona un vector perpendicular a uno dado, y finalmente `v-are-perpendicular?`, que nos dice si dos vectores son perpendiculares. Todas estas funciones son sencillas, y prácticamente una copia de sus definiciones, motivo por el que no se incluyen aquí. Lo que sí se mostrará es parte de su test de unidad, lógicamente ejecutado sobre el módulo completo:

```
(munit LINE is
   assertEqual(v-scalar-prod(v(1.0 0.0), v(0.0 1.0)), 0.0)
   assertEqual(v-scalar-prod(v(1.0 2.0), v(2.0 1.0)), 4.0)
   assertDifferent(v-norm(v(2.0 0.0)), 4.0)
   assertEqual(v-norm(v(2.0 3.0)), sqrt(13.0))
   assertDifferent(perpendicular-vector(v(1.0 2.0)), v(2.0 1.0))
   assertEqual(perpendicular-vector(v(1.0 0.0)), v(0.0 -1.0))
   assertTrue(v-are-perpendicular?(v(1.0 0.0), v(0.0 1.0)))
```

```
   assertFalse(v-are-perpendicular?(v(1.0 2.0), v(2.0 1.0)))
   assertTrue(v-are-perpendicular?(v(2.0 3.0), perpendicular-vector(v(2.0 3.0))))
endu)
```

Antes de continuar con las operaciones para rectas construiremos dos funciones opuestas, `direction-vector-line` que nos devuelve el vector director de una recta y `line-from-vector`, que recibe un vector, un punto y devuelve una recta, todo ello dentro del módulo `LINE`:

```
op direction-vector-line : Line -> Vector .
*** Devuelve el vector director de una recta
eq direction-vector-line(r(p(z11 z12) p(z21 z22))) =
   v((z21 - z11) (z22 - z12)) .

op line-from-vector : Vector Point -> Line .
*** Construye una recta mediante un vector y un punto dados.
eq line-from-vector(v(z11 z12) , p(z21 z22)) =
   r(p(z21 z22) p((z11 + z21) (z12 + z22))) .
```

Estas funciones nos ayudarán a comprobar si dos rectas son perpendiculares o a calcular una perpendicular a otra en función a un punto dado, ya que simplemente convertiremos las rectas a vectores, y con las funciones anteriores se realizará todo el proceso para después simplemente volver a convertirlos en rectas. Estas funciones son `perpendicular-line` y `are-perpendiculars` respectivamente. Volviendo a la idea original del módulo total, conseguir la geometría suficiente para imitar las construcciones con regla y compás, lo que necesitamos es hallar el punto de corte entre dos rectas. Para esto pondremos las ecuaciones en su forma implícita y resolveremos el sistema correspondiente. Comencemos entonces por encontrar los valores m y n, pendiente y desplazamiento respectivamente, de las rectas:

```
op equ-line-m : Line -> Float .
*** Dados los dos puntos que definen la recta, devuelve el valor de m,
*** para poder "construirla" de forma y = mx + n.
eq equ-line-m(r(p(z11 z12) p(z21 z22))) = (z22 - z12)/(z21 - z11) .

op equ-line-n : Line -> Float .
*** Dados los dos puntos que definen la recta, devuelve el valor de n,
*** para poder "construirla" de forma y = mx + n , teniendo cuidado
*** en las que son de la forma  x = n
eq equ-line-n(r(p(z11 z12) p(z21 z22)))= z12 -
   (z11 * equ-line-m(r(p(z11 z12) p(z21 z22)))) .

op are-equal : Line Line -> Bool .
*** comprueba si dos rectas son iguales a partir de m y n,
*** pendiente y desplazamiento, pues una recta
*** puede construirse mediante infinitos pares de puntos.
eq are-equal(r(p(z11 z12) p(z11 z22)), r(p(z11 z32) p(z11 z42))) = true .
eq are-equal(r1 , r2) = (equ-line-m(r1) == equ-line-m(r2)) and
   (equ-line-n(r1) == equ-line-n(r2)) [owise] .
```

Como se puede ver, además de las funciones antes mencionadas se ha incluido una más llamada `are-equal`, ya que la igualdad básica, `_==_`, en estos casos falla pues la misma recta puede estar definida de infinitas maneras distintas, véase: `r(p(0.0 0.0) p(1.0 1.0))` y `r(p(0.0 0.0) p(n n))` son la misma recta para todo `n` natural, pero no son sintacticamente iguales, que es lo que Maude necesita. Antes de definir los puntos de corte de dos rectas necesitaremos

dos funciones básicas para la distinción de casos, `horizontal` y `vertical` que nos indicarán, respectivamente, si la recta que le damos es horizontal o vertical. Como ya sucedió más arriba la definición de estas funciones es directa, simplemente comprobamos las coordenadas de los puntos, así que no se incluirán aquí.

Aprovechando la creación de estas funciones realizaremos la última función menor, `are-in-line` que indica si un punto pertenece a una recta comprobando la distancia del punto dado con los dos que la definen, si la suma de las distancias del punto dado a los originales es igual que la distancia entre estos el punto estará contenido en la recta. Esta función tampoco aparecerá aquí por los motivos antes mencionados, su ecuación es muy sencilla, consistiendo en una transcripción.

A continuación se muestran algunos casos del test de unidad correspondiente a las funciones para rectas.

```
(munit LINE is
   assertEqual(direction-vector-line(r(p(0.0 0.0) p(1.0 1.0))), v(1.0 1.0))
   assertDifferent(line-from-vector(v(1.0 2.0), p(0.0 0.0)),
                  r(p(1.0 2.0) p(0.0 0.0)))
   assertTrue(are-equal(line-from-vector(v(1.0 2.0), p(0.0 0.0)),
             r(p(0.0 0.0) p(1.0 2.0))))
   assertTrue(are-perpendiculars(r(p(0.0 0.0) p(0.0 1.0)),
             r(p(0.0 0.0) p(1.0 0.0))))
   assertEqual(equ-line-m(r(p(0.0 0.0) p(1.0 1.0))), 1.0)
   assertEqual(equ-line-n(r(p(1.0 1.0) p(2.0 0.0))), 2.0)
   assertTrue(are-equal(r(p(0.0 0.0) p(1.0 1.0)) , r(p(23.0 23.0) p(-1.0 -1.0))))
   assertTrue(vertical(r(p(13.0 -43.0) p(13.0 56.0))))
   assertTrue(horizontal(r(p(57.0 12.0) p(24.0 12.0))))
   assertTrue(are-in-line(p(0.0 0.0), r(p(0.0 0.0) p(1.0 1.0)))))
endu)
```

Llegamos ahora a la parte más interesante de este módulo, el punto de corte de dos rectas. Por comodidad lo primero que se hará será crear una función `cut?-r-r` que nos dirá si dos rectas se cortan y posteriormente crearemos otra que nos devuelva ese punto. En caso de que las rectas no se corten, la función que nos da el punto de corte nos devolverá un error.

```
op cut?-r-r : Line Line -> Bool .
*** comprueba si dos rectas se cortan a partir de la pendiente de estas
*** devolviendo el valor false si son paralelas y true en caso contrario.
*** El caso en que sean coincidentes debera verse de forma manual comparandolas.
ceq cut?-r-r(r1, r2) = true
if vertical(r1) and horizontal(r2) .
ceq cut?-r-r(r1, r2) = true
if vertical(r2) and horizontal(r1) .
eq cut?-r-r(r1, r2) = not (equ-line-m(r1) == equ-line-m(r2)) .
```

Para hallar el punto de corte haremos lo mencionado anteriormente, con las dos rectas en implícitas devolvemos la solución del sistema correspondiente. El método es trivial, salvo los cálculos, pero al encontrarnos en un caso general debemos hacer una distinción de casos que se corresponden con las rectas horizontales y verticales, ya que la pendiente cero e infinito da bastantes problemas.

```
op cut-point-r-r : Line Line -> Point .
*** Devuelve el punto de corte de dos rectas dadas.
ceq cut-point-r-r(r1, r2) = cut-point-r-r(r2, r1)
```

```
   if vertical(r1) .
   ceq cut-point-r-r(r1, r2) = cut-point-r-r(r2, r1)
   if horizontal(r2) .
   ceq cut-point-r-r(r1, r2) = p(x y)
   if x := cut-point-x(r1, r2) /\
      y := cut-point-y(r1, r2) .
```

Los cálculos de las funciones `cut-point-x` y `cut-point-y` se pueden encontrar en el apéndice A.1. Al igual que hemos hecho en el resto de casos comprobamos que todo funciona correctamente antes de crear el módulo incompleto. Sin embargo, a la hora de ejecutar los test nos encontramos con un único caso que no pasa, lo cual resulta extraño, que es el siguiente:

```
(munit LINE is
   assertEqual(cut-point-r-r(r(p(2.0 3.0) p(7.0 5.0)), r(p(9.3 2.4) p(7.0 5.0))),
               p(7.0 5.0))
end)
```

Cuya ejecución, que como se puede ver está forzada para dar `true`, es la siguiente:

```
1 test cases were executed.
1 failures.

assertEqual(cut-point-r-r(r(p(2.0 3.0)p(7.0 5.0)),r(p(9.3000000000000007
    2.3999999999999999)p(7.0 5.0))),p(7.0 5.0)) failed.
```

Para ver el problema calcularemos el punto por separado, pues, por como están construidas las rectas, debería devolvernos el punto `p(7.0 5.0)` .

```
Maude> red cut-point-r-r(r(p(2.0 3.0) p(7.0 5.0)), r(p(9.3 2.4) p(7.0 5.0))) .
reduce in GEO2D : cut-point-r-r(r(p(2.0 3.0) p(7.0 5.0)), r(p(
    9.3000000000000007 2.3999999999999999) p(7.0 5.0))) .
rewrites: 88 in 0ms cpu (0ms real) (~ rewrites/second)
result Point: p(7.0000000000000009 5.0000000000000009)
```

Aquí vemos el error, Maude no se comporta bien con los decimales, y aunque sea en una posición tan baja, la dieciséis, esos números no son ni siete ni cinco, así que Maude nos devuelve un fallo. Para corregir este tipo de errores incluiremos en el `fmod POINT` la función auxiliar `equal-epsilon`, que dado dos números dirá si son iguales en base a un epsilon que podremos elegir.

```
   op equal-epsilon : Float Float Float -> Bool .
   *** Comprueba si dos numeros reales son iguales en funcion de un epsilon.
   ceq equal-epsilon(z11, z12, a) = equal-epsilon(z12, z11, a)
   if z11 < z12 .
   ceq equal-epsilon(z11, z12, a) = true
   if (z11 >= z12)
      /\ a > (z11 - z12) .
   ceq equal-epsilon(z11, z12, a) = false
   if (z11 >= z12)
      /\ a <= (z11 - z12) .
```

A continuación veremos, haciendo un pequeño cambio, como el test que antes fallaba ahora nos devuelve un resultado correcto.

```
(munit LINE is
   assertEqual(cut-point-r-r(r(p(2.0 3.0) p(7.0 5.0)), r(p(9.3 2.4) p(7.0 5.0))),
               p(7.0 5.0))
   assertTrue(equal-epsilon(0.0, distance(cut-point-r-r(r(p(2.0 3.0) p(7.0 5.0)),
              r(p(9.3 2.4) p(7.0 5.0))), p(7.0 5.0)), 10.0 ^ (-5.0)))
endu)

2 test cases were executed.
1 failures.

assertEqual(cut-point-r-r(r(p(2.0 3.0)p(7.0 5.0)),r(p(9.3000000000000007
    2.3999999999999999)p(7.0 5.0))),p(7.0 5.0)) failed.
assertTrue(equal-epsilon(0.0,distance(cut-point-r-r(r(p(2.0 3.0)p(7.0 5.0)),r(
    p(9.3000000000000007 2.3999999999999999)p(7.0 5.0))),p(7.0 5.0)),1.0e+1 ^
    -5.0)) passed.
```

Se puede ver que todo ha pasado a funcionar correctamente, aunque a veces los decimales nos pueden dar problemas. De esta manera ponemos fin al módulo `fmod LINE`.

3.3. CIRCUMFERENCE

Llegamos ya al último módulo funcional para geometría en dos dimensiones, `CIRCUMFERENCE`, y es que una vez hayamos conseguido las funciones que nos den los puntos de corte entre dos circunferencias, o entre una de estas y una recta, tendremos todas las funciones necesarias para implementar las construcciones con regla y compás.

Comenzaremos con la construcción de las circunferencias, consistiendo estas en un par formado por su centro y su radio:

```
fmod CIRCUMFERENCE is

   pr LINE .
   sorts Circumference .

   op c : Point Float -> Circumference [ctor] .
   *** Define una circunferencia como un par (punto, real), correspondiendo
   *** estos con el centro y el radio.
```

De forma paralela crearemos también tres funciones muy sencillas: `circumference-center`, `circumference-radius` y `are-in-circumference`. Las dos primeras simplemente nos devuelven el centro y el radio de la circunferencia, mientras que la tercera, nos indica si un punto está en la circunferencia. Para esto simplemente calcula la distancia del punto al centro y la compara con el radio, si son iguales pertenecerá a esta y nos devolverá `true`, en caso contrario `false`:

```
   op circumference-center : Circumference -> Point .
   *** Devuelve el centro de la circunferencia
   eq circumference-center(c(p1 , z11)) = p1 .

   op circumference-radius : Circumference -> Float .
   *** Devuelve el radio de la circunferencia
   eq circumference-radius(c(p1 , z11)) = z11 .

   op are-in-circumference : Point Circumference -> Bool .
```

```
*** Comprueba si un punto esta en una circunferencia,
***  comprobando su distanciaal centro con el radio.
eq are-in-circumference(p1 ,c(p2 , z11)) = distance(p1 , p2) == z11 .
```

Mientras que con las rectas hubo variedad de funciones, en el caso de las circunferencias nos centraremos en hallar los puntos de corte comenzando por el caso de recta y circunferencia. Comenzaremos creando una función que nos dirá si existe el punto de corte, siguiendo los siguientes pasos: trazamos una recta perpendicular a la dada que pase por el centro de la circunferencia y hallamos el punto de corte. A continuación calculamos la distancia de ese punto al centro, si es menor o igual que el radio significará que la recta y la circunferencia se cortan, no habrá corte en caso contrario. Veamos la función:

```
op cut?-r-c : Line Circumference -> Bool .
*** Comprueba si una recta y una circunferencia se cortan mediante la comparacion
*** de la distancia de la recta al centro de la circunferencia.
eq cut?-r-c(r1 , c(p1 , z11)) = distance(cut-point-r-r(r1 ,
   perpendicular-line(r1 , p1)) , p1) <= z11 .
```

Una vez hecho esto tenemos que ver cómo conseguir los puntos de corte. El primer problema que nos encontramos es el número, ya que pueden cortarse en uno o dos puntos. Para ello implementaremos en el `fmod POINT` el tipo `List` :

```
sorts Point List .
subsort Point < List .

op p(__) : Float Float -> Point [ctor] .
*** Define un punto como un par de numeros reales

op mt : -> List [ctor] .
op __ : List List -> List [ctor assoc id: mt] .
```

La definición que hacemos aquí de las listas es idéntica la que ya hicimos en el primer capítulo, con la salvedad de que ahora son listas de puntos. Las funciones definidas son la mismas que la otra vez, con solamente dos nuevas, `first-element` y `delete-first-element` que respectivamente nos dan el primer elemento de una lista, o nos dan esta sin el primer elemento.

Sin embargo estas no son las únicas funciones nuevas que añadiremos al módulo, dada la forma de las ecuaciones tanto de la recta como de la circunferencia necesitaremos resolver llegado el momento ecuaciones de segundo grado. Así pués implementamos las funciones correspondientes en `POINT`:

```
op first-degree-equation : Float Float -> Float .
*** Devuelve la solucion de una ecuacion de primer grado.
*** Recibe los valores correspondientes a ax + b.
eq first-degree-equation(a, b) = - b / a .

op second-degree-equation-1 : Float Float Float -> Float .
*** Devuelve el valor de una ecuacion de segundo grado en el caso +
*** Recibe los valores correspondientes a ax^2 + bx + c.
eq second-degree-equation-1(0.0 , b , c) = first-degree-equation(b, c) .
eq second-degree-equation-1(a , b , c) =
   (- b + sqrt(b ^ 2.0 - (4.0 * a * c))) / (2.0 * a) .
```

```
op second-degree-equation-2 : Float Float Float -> Float .
*** Devuelve el valor de una ecuacion de segundo grado en el caso -
*** Recibe los valores correspondientes a ax^2 + bx + c.
eq second-degree-equation-2(0.0 , b , c) = first-degree-equation(b, c) .
eq second-degree-equation-2(a , b , c) =
   (- b - sqrt(b ^ 2.0 - (4.0 * a * c))) / (2.0 * a) .
```

Terminadas todas las funciones auxiliares que necesitaremos solo queda implementar las funciones de corte en el módulo `CIRCUMFERENCE`. Para ello volveremos a hacer una distinción de casos, siendo esta vez tres. Los dos primeros son aquellos en los que las rectas son verticales u horizontales, ya que eso nos da automáticamente uno de los valores de los puntos. Sin embargo, como puede verse en el programa, no he hecho uso de las funciones definidas en el `LINE`, sino que se ven si son horizontales o verticales por los puntos que las forman. Por otro lado tenemos el caso general, que calcula por separado los dos puntos de corte y los devuelve en forma de lista. Podemos encontrar los cálculos referentes a estas funciones aquí A.2. Veamos la función principal:

```
op cut-point-r-c : Line Circumference -> List .
*** Dados una circunferencia y un punto devuelve una lista
*** con los dos puntos de corte pudiendo ser estos iguales.
ceq cut-point-r-c(r(p(z11 z12) p(z11 z22)), c(p(z31 z32), z41)) =
    p(z11 Aux-A) p(z11 Aux-B)
*** caso vertical
if Aux-A := second-degree-equation-1(1.0, (-2.0 * z32), (z32 ^ 2.0) +
   ((z11 - z31) ^ 2.0) - (z41 ^ 2.0)) /\
   Aux-B := second-degree-equation-2(1.0, (-2.0 * z32), (z32 ^ 2.0) +
   ((z11 - z31) ^ 2.0) - (z41 ^ 2.0)) .
ceq cut-point-r-c(r(p(z11 z12) p(z21 z12)), c(p(z31 z32), z41)) =
    p(Aux-A z12) p(Aux-B z12)
*** caso horizontal
if Aux-A := second-degree-equation-1(1.0, (-2.0 * z31), (z31 ^ 2.0) +
   ((z21 - z32) ^ 2.0) - (z41 ^ 2.0)) /\
   Aux-B := second-degree-equation-2(1.0, (-2.0 * z31), (z31 ^ 2.0) +
   ((z21 - z32) ^ 2.0) - (z41 ^ 2.0)) .
eq cut-point-r-c(r1, c1) = cut-point-r-c-1(r1, c1) cut-point-r-c-2(r1, c1) .
```

Las dos funciones auxiliares `cut-point-r-c-1` y `cut-point-r-c-2` se omiten por comodidad. Estas se encargan de calcular los puntos de corte sustituyendo la recta en la circunferencia. Concluida ya esta función nos ponemos con la última operación que necesitaremos, calcular los puntos de corte de dos circunferencias. Lo primero que haremos será construir la función que nos indique si las dos figuras se cortan, como ya hicimos en los otros casos. Esta vez la idea de la función resulta mucho más sencilla que antes, simplemente comprobamos la distancia entre los centros y vemos si es menor o igual que la suma de los radios.

```
op cut?-c-c : Circumference Circumference -> Bool .
*** Comprueba si dos circunferencias se cortan en base a
*** la distancia de los centros.
ceq cut?-c-c(c1, c2) = distance(circumference-center(c1),
    circumference-center(c2)) <= (circumference-radius(c1)
                                + circumference-radius(c2))
if not (circumference-center(c1) == circumference-center(c2)) .
ceq cut?-c-c(c1, c2) = circumference-radius(c1) == circumference-radius(c2)
if (circumference-center(c1) == circumference-center(c2)) .
```

Sin embargo hallar los puntos de corte sí resulta más complicado, y es que si las dos circunferencias poseen centro con la coordenada x igual, nos surgirá más adelante una división entre cero, así que en ese caso iremos por otro camino, que dependerá de que sean distintas las coordenadas y. Esto se debe a que haremos una simetría respecto de la recta $y = x$ para hallar los puntos de corte, y luego los traspondremos, lo cual también corresponde con hacer una simetría. Este proceso simplemente se hace para evitar tener que hacer una nueva distinción de casos, pues los calculos serían totalmente análogos:

```
op cut-point-c-c : Circumference Circumference -> List .
ceq cut-point-c-c(c(p(z11 z12) , z31) , c(p(z21 z22) , z32)) =
    cut-point-circumferences-x-dis(c(p(z11 z12) , z31) , c(p(z21 z22) , z32))
if not (z11 == z21) .
ceq cut-point-c-c(c(p(z11 z12) , z31) , c(p(z21 z22) , z32)) =
    cut-point-circumferences-y-dis(c(p(z11 z12) , z31) , c(p(z21 z22) , z32))
if not (z12 == z22) .
ceq cut-point-c-c(c(p(z11 z12) , z31) , c(p(z21 z22) , z32)) = mt
if p(z11 z12) = p(z21 z22) /\ not(z31 == z32) .

op cut-point-circumferences-x-dis : Circumference Circumference -> List .
*** da una lista con dos puntos, que serian los puntos de corte de las
***  dos circunferencias.
*** solo funciona si los centros no tienen la misma X.
ceq cut-point-circumferences-x-dis(c1 , c2) = p1 p2
if p1 := cut-point-circumferences-1(c1 , c2)
/\ p2 := cut-point-circumferences-2(c1 , c2) .
```

Ahora simplemente damos las funciones que nos devuelven los puntos propiamente dichos y listo. Como ocurría con los demas puntos de corte estos cálculos no son triviales, y pueden verse aquí A.3.

```
op cut-point-circumferences-1 : Circumference Circumference -> Point .
*** halla el primer punto de corte de las dos circunferencias.
ceq cut-point-circumferences-1(c(p(z11 z12) , z31) , c(p(z21 z22) , z32)) =
    p((Aux-A - (second-degree-equation-1(Aux-C , Aux-D , Aux-E) * Aux-B))
       second-degree-equation-1(Aux-C , Aux-D , Aux-E))
if Aux-A := (z31 ^ 2.0 - z32 ^ 2.0 + (z21 ^ 2.0 + z22 ^ 2.0) -
            (z11 ^ 2.0 + z12 ^ 2.0)) / (2.0 * z21 - 2.0 * z11)
/\ Aux-B := (z22 - z12) / (z21 - z11)
/\ Aux-C := (Aux-B) ^ 2.0 + 1.0
/\ Aux-D := 2.0 * Aux-B * (z11 - Aux-A) - 2.0 * z12
/\ Aux-E := Aux-A * (Aux-A - 2.0 * z11) - z11 ^ 2.0 + z12 ^ 2.0 - z31 ^ 2.0 .

op cut-point-circumferences-2 : Circumference Circumference -> Point .
*** halla el segundo punto de corte de las dos circumferences.
ceq cut-point-circumferences-2(c(p(z11 z12) , z31) , c(p(z21 z22) , z32)) =
    p((Aux-A - (second-degree-equation-2(Aux-C , Aux-D , Aux-E) * Aux-B))
       second-degree-equation-2(Aux-C , Aux-D , Aux-E))
if Aux-A := (z31 ^ 2.0 - z32 ^ 2.0 + (z21 ^ 2.0 + z22 ^ 2.0) -
            (z11 ^ 2.0 + z12 ^ 2.0)) / (2.0 * z21 - 2.0 * z11)
/\ Aux-B := (z22 - z12) / (z21 - z11)
/\ Aux-C := (Aux-B) ^ 2.0 + 1.0
/\ Aux-D := 2.0 * Aux-B * (z11 - Aux-A) - 2.0 * z12
/\ Aux-E := Aux-A * (Aux-A - 2.0 * z11) - z11 ^ 2.0 + z12 ^ 2.0 - z31 ^ 2.0 .
```

Terminadas ya todas las funciones referidas a las circunferencias se procede a la creación del test correspondiente, el cual puede encontrarse en el archivo correspondiente del repositorio ya que no aportan ninguna información importante. Con esto damos fin al módulo de circunferencias.

3.4. GEO2D

Habiendo concluido los módulos `POINT`, `LINE` y `CIRCUNFERENCE` nos encontramos con que hay puntos de corte entre muchos tipos de figuras, pero según sean estas de un tipo u otro tendremos que usar distintas funciones. Por ello creamos un último módulo llamado GEO2D, que solo incluirá una función llamada `cut-point`, que dadas dos figuras cuales quiera, devolverá sus puntos de corte.

Para ello empezaremos por crear un nuevo tipo de funciones, llamado `Figure`, que simplemente contendrá a los demas tipos, haciendo así que nos sea totalmente indiferente trabajar con una u otra figura llegado el momento.

```
fmod GEO2D is

   pr CIRCUMFERENCE .
   sort Figure .
   subsort Point Line Circumference < Figure .

   op cut-point : Figure Figure -> List [comm] .
   *** Comprueba si dos figuras se cortan, y devuelve una lista con
   *** los puntos de corte
   ...
endfm
```

Capítulo 4

Módulo interactivo

A lo largo de este capítulo explicaremos la creación del módulo encargado de la entrada y la salida de datos, y de los diccionarios de figuras, que serán la estructura de datos que utilizaremos para almacenar la información.

4.1. Diccionarios

Dando por concluido los módulos referidos a la geometría en dos dimensiones comenzamos con el de entrada/salida. Queremos que este tenga una interfaz simple para crear las figuras, del tipo: "la circunferencia c1 con centro el punto p y radio z". Así pues necesitaremos crear una estructura de datos que se encargue de almacenar toda la información. Para esto podríamos crear simplemente una lista, como ya hemos hecho otras veces, pero resulta más natural, al constar las figuras de nombre, que hagamos uso directamente de un diccionario.

El diccionario deberá contar con unas estructuras internas más sencillas que llamaremos entradas (`entry`), las cuales llevarán por un lado el nombre, y por otro la figura asociada. Para los tipos de las figuras no tenemos ningun problema, ya creamos el tipo `Figure` en el módulo `GEO2D`. Para los nombres por el contrario no tenemos ningún tipo creado por nosotros que nos sirva, así que haremos uso de uno incluido en Core Maude, los `Qid`. Con esto ya decidido veamos la creación de los diccionarios:

```
fmod DICC-FIGURES is
   pr GEO2D .
   pr QID .

   sorts Dic Entry .
   subsorts Entry < Dic .

   op _->_ : Qid Figure -> Entry [ctor] .
   op mtD : -> Dic [ctor] .
   op _._ : Dic Dic -> Dic [ctor assoc comm id: mtD] .
```

Como se puede ver la definición resulta natural, `mtD` es el diccionario vacío y `_._` la operación que nos permite unirlos, siendo el vacío su elemento neutro. Los `Entry` por otro lado se definen como ya habíamos dicho anteriormentes, uniendo sus dos términos mediante una flecha.

A continuación definimos las operaciones de los diccionaros que harán uso sobre todo de la commutatividad de la unión para facilitar su implementación. La primera función que crearemos será aquella que dado un diccionario y un `Qid` nos devuelva la figura correspondiente, seguida de aquella que nos indica recibiendo los mismos datos si el diccionario contiene alguna figura con ese identificador. Estas funciones serían :

```
op _[_] : Dic Qid ~> Figure .
*** Dado un Qid devuelve la figura
*** correspondiente.
eq (D . Q -> F)[Q] = F .
```

```
op contains? : Dic Qid -> Bool .
*** Dado un Qid indica si el diccionario
*** tiene una entreda que le corresponda.
eq contains?((Q -> F) . D, Q) = true .
eq contains?(D, Q) = false [owise] .
```

Continuando con estas operaciones básicas se procederá a crear una que nos permita actualizar el diccionario. Esta función se encargará además de evitar las repeticiones, tanto en los identificadores como en las figuras, y en caso de que la información no estuviese ya en él simplemente la incluye. Esta función quedaría entonces como sigue:

```
op _[_/_] : Dic Qid Figure -> Dic .
eq (D . Q -> F)[Q / F'] = D . Q -> F' .
eq (D . Q -> F)[Q' / F] = D . Q -> F .
ceq (D . Q -> r1)[Q' / r2] = D . Q -> r1
  if are-equal(r1, r2) .
eq D[Q / F] = D . Q -> F [owise] .
```

Como se puede ver presenta un caso diferenciado, que es aquel que se refiere a las rectas, que como ya se explicó en el apartado correspondiente, debido a la forma que tienen estas. El resto de la función es sencillo; el primer caso cubre una actualización, el segundo, junto con el de las rectas, evita que introduzcamos de nuevo una figura pero con distinto nombre. El último simplemente cubre el caso de que no estuviese y lo añade directamente.

Para concluir esta sección crearemos dos funciones, la primera de ellas (`_in_`) será análoga a `contains?` pero en este caso recibirá la figura en vez del nombre, la segunda (`name`) será análoga a aquella que nos daba una figura dado su nombre. Sin embargo en este caso sí cubrirá el caso en que no esté, pues lo necesitamos a la hora de mostrar la información por pantalla. Para ello haremos uso de un nuevo tipo `QidList`, que se corresponde con las listas de `Qid`, de él tomaremos un único elemento, `nil`, la lista vacía, que usaremos para no devolver nada. Mostramos a continuación estas dos funciones:

```
op _in_ : Figure Dic -> Bool .
*** Indica si la figura pertenece al
*** diccionario.
eq F in D . (Q -> F) = true .
eq F in D = false [owise] .
```

```
op name : Dic Figure -> QidList .
*** Dado un diccionnario y una figura
*** devuelve el identificador de esta.
eq name(D . Q -> F, F) = Q .
eq name(D , F) = nil [owise] .
```

A continuación daremos algunos casos del test de unidad para diccionarios para ver claramente su sintaxis:

```
(munit DICC-FIGURES is
   assertEqual('p1 -> p(0.0 0.0) ['p1], p(0.0 0.0))
   assertTrue(contains?(('p1 -> p(0.0 0.0)) . ('p2 -> p(1.0 1.0)), 'p1))
   assertEqual('p1 -> p(0.0 0.0) ['p1 / p(1.0 1.0)], 'p1 -> p(1.0 1.0))
   assertFalse(p(0.0 1.0) in ('p1 -> p(0.0 0.0)) . ('p2 -> p(1.0 1.0)))
   assertEqual(name('p -> p(0.0 0.0), p(0.0 0.0)), 'p)
```

4.1.1. Diccionario de figuras

Simplemente con estos diccionarios podríamos dar por creada la estructura de datos. Sin embargo, aunque son totalmente funcionales, carecen de orden, ya que los puntos, rectas y circunferencias se encuentran totalmente mezclados. Así pues construiremos un nuevo tipo `DicFigures` que se corresponde con un conjunto de tres diccionarios, estando cada uno de ellos destinado a un tipo de figura concreta. Su definición sería entonces la que sigue:

```
sorts Dic Entry DicFigures .
subsorts Entry < Dic .

op [_ , _ , _] : Dic Dic Dic -> DicFigures [ctor] .
```

Las funciones que definiremos para estos nuevos diccionarios serán las mismas que para los dados anteriormente y utilizarán los mismos comandos para mayor comodidad. Un ejemplo de estas serían las siguientes:

```
op _._ : DicFigures Entry -> DicFigures .
eq [Dp , Dr , Dc] . Q -> p1 = [(Dp . Q -> p1) , Dr , Dc] .
eq [Dp , Dr , Dc] . Q -> r1 = [Dp , (Dr . Q -> r1) , Dc] .
eq [Dp , Dr , Dc] . Q -> c1 = [Dp , Dr , (Dc . Q -> c1)] .

op _[_] : DicFigures Qid ~> Figure .
eq [Dp . Q -> p1 , Dr , Dc][Q] = p1 .
eq [Dp , Dr . Q -> r1 , Dc][Q] = r1 .
eq [Dp , Dr , Dc . Q -> c1][Q] = c1 .
```

Los casos de test serían entonces totalmente análogos, pero al igual que antes incluiremos alguno como ejemplo:

```
assertEqual(([mtD, mtD, mtD] ['p1 / p(1.0 1.0)]), [('p1 -> p(1.0 1.0)), mtD, mtD])
assertEqual(([mtD, mtD, mtD] ['r1 / r(p(1.0 1.0) p(0.0 0.0))]),
             [mtD, ('r1 -> r(p(1.0 1.0) p(0.0 0.0))), mtD])
assertEqual(([mtD, mtD, mtD] ['c1 / c(p(1.0 1.0), 1.0)]),
             [mtD, mtD, ('c1 -> c(p(1.0 1.0), 1.0))])
```

4.2. Módulo interactivo

Una vez terminado el módulo referente a las estructuras de datos comenzaremos con el de entrada/salida. Este se encontrará dividido en cinco módulos distintos, situados en el mismo archivo, dedicándose cada uno a aspecto distinto de lo que queremos implementar. Para la construcción de estos módulos será necesario importar Full Maude, que importaremos al comienzo del archivo. Lo que sí necesitaremos importar dentro del módulo será `FULL-MAUDE-SIGN` pero como este sí lo modicaremos al incluir nuevas funciones de entrada, lo importamos con el comando `inc`.

En este primer módulo que crearemos daremos los comandos que más adelante utilizaremos para la entrada de datos. Por esto hay que tener cuidado con los tipos, ya que mediante inputs Maude recibe `Token` y `Bubble`, los cuales tendremos que modificar despues para obtener la información requerida. Como de momento solo nos interesa la creación de figuras solo crearemos los comandos de estas:

```
fmod GEO-SIGNATURE is
  inc FULL-MAUDE-SIGN .

  sorts @GeoCommand@ .
  subsort @GeoCommand@ < @Input@ .

  op point_as_ : @Bubble@ @Token@ -> @GeoCommand@ [ctor] .
  op circumference_in_with‘radius_ : @Token@ @Token@ @Bubble@
                                     -> @GeoCommand@ [ctor] .
  op line_from_to_ : @Token@ @Token@ @Token@ -> @GeoCommand@ [ctor] .

endfm
```

Como se puede ver la definición es análoga a la de cualquier otra en Maude, con la salvedad de que los tipos deben darse entre `@`.

Tal y como estan dados los comandos veamos cómo se dan los comandos: los puntos se escribirían `point p(0.0 0.0) as p1`, las circunferencias `circumference c1 in p1 with radius 1.0`, y finalmente las rectas como `line r1 from p1 to p2`. Como se puede ver todas las figuras menos los puntos depende de tener ya otras creadas. Esto es así para representar la necesidad de conocer otras figuras para construir unas nuevas.

Una vez concluido este módulo nos encargaremos de hacer que Maude entienda lo que queremos hacer, para ello crearemos un módulo de sistema llamado `GEO-DATABASE-HANDLING` en el que ir recogiendo las distintas reglas que se encargarán de interpretar la información. Estas reglas serán analogas a la siguiente:

```
 crl [point] :
     < O : MUDC | input : ('point_as_[T1 , T2]), dic : D,
                  output : nil, AtS >
  => < O : MUDC | input : nilTermList, dic : (D [Q / F]),
                  output : ('Punto 'almacenado names) , AtS >
  if Q -> F := processPoint(T1 , T2) /\
     names := mod-elements(D [Q / F], Q -> F) .
```

Empezaremos explicando el significado de lo que se acaba de mostrar, la regla nos da una transición entre dos estados del sistema, de los cuales los elementos que nos interesan son los tres centrales, véase, el input, el dic, y el output. El input se corresponde con los comandos puestos arriba, donde T1 y T2 son el punto y su nombre respectivamente. El dic es logicamente el diccionario que llevamos en todo momento como estructura de datos donde almacenaremos todas las figuras y sería un `DicFigures`. Finalmente el output será la información mostrada por pantalla, en nuestro caso el nombre del punto almacenado.

El funcionamiento de la regla comieza calculando `processPoint(T1, T2)` que nos devolverá una entrada de la forma `Q ->F`. Una vez hecho esto el diccionario se actualizará mediante la función correspondiente (`dic : (D [Q / F])`) con la nueva información. Finalmente indica mediante un output que el punto ha sido almacenado, indicando eso sí solamente el nombre. Para esto último se realizará un llamamiento a la función `mod-elements` que nos dará los nombres de las figuras que han sido incluidas en el diccionario, pero en el caso de ya pertenecer a él, nos dará el nombre original de estas, véase, con el que se incluyeron la primera vez. Esto es así para evitar dar información incorrecta por pantalla. La función `mod-elements`, que no se mostrará aquí, se puede encontrar en el módulo auxiliar del que hablaremos a continuación.

Como se ha podido ver en la regla anterior, se hace un llamamiento a la función `processPoint`, ésta se encuentra en un módulo funcional auxiliar llamado `GEO-COMMAND-PROCESSING` que se encargará de estas funciones y otras auxiliares como la mencionada antes. Como hemos dado

de ejemplo la regla para los puntos veremos a continuación la función `processPoint`, que se encargará de convertir los datos de entrada a los tipos que necesitamos mediante otras dos:

```
op processPoint : Term Term -> Entry .
eq processPoint('bubble[T], 'token[NP]) = downQid(NP) -> processPointAux(T) .

op processPointAux : Term -> Point .
ceq processPointAux('__[T, T']) = qid2point(Q, Q')
 if Q := downQid(T) /\
    Q' := downQid(T') .

op qid2point : Qid Qid -> Point .
ceq qid2point(Q, Q') = p(F F')
 if S := string(Q) /\
    S' := string(Q') /\
    F := float(S) /\
    F' := float(S') .
```

Veamos lo que hacen las funciones anteriores. `processPoint` se encarga de extraer el nombre de uno de los términos, en este caso de `'token[NP]`, y dar la entrada al diccionario utilizando la función `processPointAux` que dado el término `p(_ _)` lo descompone, y a través de `qid2point` lo convierte en un punto como los que definimos atrásss.

Nos quedaría por tanto explicar las reglas para las rectas y las circunferencias y sus `process`, pero se omitirán al ser análogas a la que acabamos de dar.

Dadas las construcciones para las tres figuras procederemos a calcular los puntos de corte, para lo cual extenderemos el módulo `GEO-SIGNATURE` con un nuevo comando:

```
op cut‘point_and_of_and_ : @Token@ @Token@ @Token@ @Token@ -> @GeoCommand@ [ctor] .
```

Como se puede ver recibe cuatro elementos: los dos primeros corresponden con los nombres de los posibles puntos de corte, recordemos que dos circunferencias se pueden cortar en dos puntos, mientras que los dos últimos son los nombres de las figuras cuyos puntos de corte estamos calculando. La regla correspondiente sería la siguiente:

```
crl [cut1] :
    < O : MUDC | input : ('cut‘point_and_of_and_[T1, T2, 'token[T3],
                  'token[T4]]), dic : D, output : nil, AtS >
 => < O : MUDC | input : nilTermList, dic : (D U [D' , mtD , mtD]),
                  output : ('Puntos 'almacenados names) , AtS >
 if D' := processCutPoint(T1 , T2 , D[downQid(T3)] , D[downQid(T4)]) /\
    names := mod-elements(D U [D' , mtD , mtD], D').
```

Aunque la regla es similar a las ya dadas debe llamarnos la atención `D U [D' , mtD , mtD]` dado que no tenemos ninguna función de los diccionarios que sea así. Esta es simplemente una unión de estos, y se puede ver en el módulo correspondiente. Lo verdaderamente importante es el motivo de que este ahí. En todas las reglas anteriores sabíamos que solo se añadía una figura al diccionario, y por tanto podíamos hacerlo manualmente; el problema es que ahora no sabemos cuántas serán, ya que las figuras pueden no cortarse, ser dos rectas, o dos circunferencias así que se procede a simplemente unir los diccionarios para evitar problemas o la creación de más casos.

Como se puede ver en `processCutPoint(T1, T2, D[downQid(T3)], D[downQid(T4)])` la función `processCutPoint` no será demasiado complicada, pues recibe directamente figuras sobre las que hacer los cálculos:

```
op processCutPoint : Term Term Figure Figure -> Dic .
ceq processCutPoint('token[T] , 'token[T'] , f1 , f2) =
    processCutPointAux(p1, p2, f1, f2, 1entrada,2entrada)
 if p1 := first-element(cut-point(f1 , f2))
 /\ p2 := first-element(delete-first-element(cut-point(f1 , f2)))
 /\ 1entrada := downQid(T) -> p1
 /\ 2entrada := downQid(T') -> p2 .
```

p1 y p2 se corresponderán entonces con los dos puntos puntos de corte, y 1entrada y 2entrada con los las entradas para el diccionario. La función processCutPointAux sería entonces lo más llamativo, pero la verdad es que lo único que hace es comprobar que no devuelven la misma figura, evitando asi que almacenemos el mismo punto dos veces y mostremos por pantalla el nombre de dos figuras iguales, o de una que no existe, pues p2 puede ser la lista vacía de puntos.

Dados los tres módulos que acabamos de crear tenemos prácticamente todo lo necesario para la entrada y salida de datos. Solo nos quedaría crear un módulo para incluir los comandos que hemos creado en la gramática de Full Maude, y un segundo que se encargará da darnos un entorno. Para comenzar se encargará de dar unos valores iniciales sobre los que trabajar, el diccionario de figuras vacío, dic : [mtD, mtD, mtD], así como inicializar el bucle dentro del que se realizan las acciones, en nuestro caso init-geo, y finalmente controlar que la entrada y la salida funcionan tal y como queremos con distintos casos de errores. Estos dos módulos son fmod META-GEO-SIGN y mod IOGEO2 respectivamente.

Veamos ahora cómo funciona la entrada y la salida de geoIO.maude con casos sencillos:

```
Maude> load geoIO.maude

    Full Maude 2.7 March 10th 2015

GEO: GEO2D a unit testing framework for Maude.
(March 7th, 2017)

Maude> (point 0.0 0.0 as p1)
Punto almacenado p1
Maude> (point 0.0 1.0 as p2)
Punto almacenado p2
Maude> (line r from p1 to p2)
Recta almacenada r
Maude> (circumference c in p1 with radius 1.0)
Circunferencia almacenada c
```

Parece que todo va bien, pero solo con esta información no es seguro, así que utilizaremos cont para comprobar la información almacenada en el diccionario.

```
Maude> cont .
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
result System: [(nil).TypeList,< o : GEODatabase | db :
db(...),input : nilTermList,output : nil,default : 'CONVERSION,dic : [('p1 ->
    p(0.0 0.0)) . 'p2 -> p(0.0 1.0),'r -> r(p(0.0 0.0) p(0.0 1.0)),'c -> c(p(
    0.0 0.0), 1.0)] >,'Circunferencia 'almacenada 'c]
```

Como se puede ver nos da bastante información, pero lo que nos interesa es el diccionario dic : [('p1 -> p(0.0 0.0)) . 'p2 -> p(1.0 1.0),'r -> r(p(0.0 0.0) p(1.0 1.0)) ,'c -> c(p(0.0 0.0), 1.0)] en el que podemos ver que toda la información se ha almacenado correctamente tal y como queríamos. Ahora solo nos quedaría ver los puntos de corte y habríamos terminado:

```
Maude> (cut point p3 and p4 of c and r)
Puntos almacenados p2 p3
Maude> cont .
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
result System: [(nil).TypeList,< o : GEODatabase | db :
db(...),input : nilTermList,output : nil,default : 'CONVERSION,dic : [('p1 ->
    p(0.0 0.0)) . ('p2 -> p(0.0 1.0)) . 'p3 -> p(0.0 -1.0),'r -> r(p(0.0 0.0)
    p(0.0 1.0)),'c -> c(p(0.0 0.0), 1.0)] >,'Puntos 'almacenados 'p2 'p3]
```

De nuevo se puede ver que funciona perfectamente, ya que los puntos de corte de las dos figuras son `p(0.0 1.0)` y `p(0.0 -1.0)`, pero como uno ya estaba en el diccionario no se hace nada, aunque se indica que ha sido añadido para dejar claro que había dos puntos de corte.

A continuación veremos algunos casos del test de unidad para las funciones de estos módulos. Al estar trabajando en Full Maude, utilizaremos dos nuevos comandos, además de todos los que vimos para Core Maude, que explicaremos a continuación:

```
(munit IOGEO2D is
  loop(init-geo)
```

El primero de ellos es `loop(_)` que se encarga de reiniciar el bucle donde se ejecuta la entrada y salida de datos.

```
  command(point 0.0 0.0 as p1)
  assertTrue(contains?(@dic, 'p1))
```

El segundo de ellos es `command(_)`, que nos permite usar los comandos ya creados en el bucle que acabamos de reiniciar. Además, se puede ver dentro del `assertTrue` que para llamar al diccionario necesitamos escribir `@dic` para indicar que nos referimos al creado en `init-geo`. Finalmente el test se cierra con `endu`:

```
endu)
```

Simplemente con estos comandos podríamos dar por terminado esta parte y con ella el proyecto. Sin embargo, para dar unas construcciones de circunferencias más realistas daremos un nuevo comando que en vez de recibir un número real tome como radio dos puntos:

```
   op circumference_in_with‘radius‘distance‘from_to_ :
      @Token@ @Token@ @Token@ @Token@ -> @GeoCommand@ [ctor] .
```

Las funciones asociadas a este comando así como la regla correspondiente son análogas a las vistas para los anteriores, así que no la mostraremos aquí.

De forma paralela a la creación de este comando se crea una nueva función que se encargará de dar nombre automáticamente a las puntos de corte calculados, y así simplificar los comandos. Esa función es la siguiente:

```
 op new-name : Qid Qid Qid DicFigures -> Qid .
 ceq new-name(name1, name2, Q, DF) = new-name(Q', qid(string(1.0)), Q, DF)
   if Q' := qid(string(name1) + string(name2)) /\
      Q' == Q  .
 ceq new-name(name1, name2, Q, DF) = new-name(Q', qid(string(1.0)), Q, DF)
   if Q' := qid(string(name1) + string(name2)) /\
      contains?(DF, Q') .
 ceq new-name(name1, name2, Q, DF) = Q'
   if Q' := qid(string(name1) + string(name2)) .
```

La función recibe los nombres de los dos elementos que la forman, como por ejemplo dos rectas, un tercer nombre que servirá, en el caso de tener dos puntos de corte, para evitar que los dos se llamen igual, y finalmente el diccionario total con todas las figuras. La función se encargaría entonces de sumar los nombres de los dos elementos que le forman teniendo cuidado de que no exista ninguna figura anterior con el mismo nombre. El caso de los puntos de corte es distinto ya que los puntos se nombran antes de añadirlos, o actualizarlos, al diccionario.

Como se puede ver esta función resulta entonces muy útil saliéndose de su propósito base, y es que dando como tercer `Qid 'none` podemos conseguir que ninguno de los comandos a excepción de los números necesiten nombrarse. Estos últimos comandos son:

```
op circumference‘in_with‘radius‘distance‘from_to_ : @Token@ @Token@ @Token@ ->
   @GeoCommand@ [ctor] .
op line‘from_to_ : @Token@ @Token@ -> @GeoCommand@ [ctor] .
op cut‘point‘of_and_ : @Token@ @Token@ -> @GeoCommand@ [ctor] .
```

Y la regla de los puntos de corte queda como sigue:

```
crl [cut2] :
    < O : MUDC | input : ('cut‘point‘of_and_['token[T1], 'token[T2]]), dic : D,
                 output : nil, AtS >
 => < O : MUDC | input : nilTermList, dic : (D U [D' , mtD , mtD]),
                 output : ('Puntos 'almacenados names) , AtS >
 if Q := downQid(T1) /\
   Q' := downQid(T2) /\
   name1 := new-name(Q, Q', 'none, D) /\
   name2 := new-name(Q', Q, name1, D) /\
   D' := processCutPoint('token[upTerm(name1)], 'token[upTerm(name2)], D[Q], D[Q'])
   /\ names := mod-elements(D U [D' , mtD , mtD], D') .
```

El resto de reglas son análogas y se pueden encontrar en el módulo completo disponible en el repositorio, junto con los respectivos casos de test.

Con estas últimas reglas damos entonces por terminado este capítulo y el proyecto, con el que hemos conseguido emular perfectamente en Maude la geometría euclídea con regla y compás mientras desarrollábamos una buena cantidad de test de unidad y los módulos de aprendizaje.

Capítulo 5

Conclusiones y trabajo futuro

El presente proyecto nos ha permitido implementar en Maude un entorno interactivo muy sencillo de utilizar que nos permite trabajar con la geometría Euclídea clásica con regla y compás. Pero no solo eso, sino que nos ha proporcionado una gran cantidad de módulos distintos con sus correspondientes test de unidad, el principal objetivo de este trabajo de fín de carrera. Por otro lado hemos podido desarrollar los módulos de aprendizaje, disponibles en el repositorio, los cuales facilitarán la tarea de aquellos que decidan iniciarse en este nuevo lenguaje mediante númerosos casos.

Sin embargo, y sin contradecir lo anterior, este proyecto no se encuentra cerrado, sino que se presta con mucha facilidad a seguir expandiéndolo. Este trabajo futuro dependerá del camino que se quiera seguir; las opciones más sencillas pasarían por la expansión de esta geometría a las tres dimensiones, pudiendo trazar en ella planos y esferas. Sin embargo esta no es la única posibilidad, pudiendo también dirigirse a un ambiente más cercano a la geometría diferencial mediante la creación de curvas y posteriormente superficies. Todas estas opciones nos permitirían continuar con la creación de Test de unidad. Sin embargo, se podría optar también por demostar la corrección y la completitud de los módulos y aprovechar las reglas dadas para crear un constructor automático de figuras con regla y compás.

Apéndice A

Puntos de corte

A.1. Punto de corte entre dos rectas

A continuación se mostrarán los cálculos referentes al punto de corte entre dos rectas dadas de la forma $y = mx + n$ y $y = m'x + n'$:

Igualamos una a la otra por la coordenada y y despejamos x:

$$mx + n = m'x + n'$$
$$(m - m')x = n' - n$$
$$x = \frac{(n' - n)}{(m - m')}$$

Con la x calculada sustituimos en la ecuación de una de las rectas y terminamos:

$$y = mx + n$$
$$y = \frac{m((n' - n)}{(m - m')) + n}$$

A.2. Punto de corte entre una recta y una circunferencia

A continuación se mostrarán los cálculos referentes al punto de corte entre una recta y una circunferencia dadas de la forma: $y = mx + n$ y $(x - x')^2 + (y - y')^2 = r^2$ siendo (x', y') el centro de la circunferencia y r su radio:

Empezamos desarrollando los cuadrados en la ecuación de la circunferencia:

$$(x - x')^2 + (y - y')^2 = r^2$$
$$x^2 + x'^2 - 2xx' + y^2 + y'^2 - 2yy' = r^2$$
$$x^2 - 2xx' + y^2 + 2yy' = r^2 - x'^2 - y'^2$$

Sustituimos ahora la recta en la ecuación:

$$x^2 - 2xx' + (mx + n)^2 + 2(mx + n)y' = r^2 - x'^2 - y'^2$$
$$x^2 - 2xx' + mx^2 + n^2 + 2mnx + 2(mx + n)y' = r^2 - x'^2 - y'^2$$

Agrupamos las x :

$$x^2 + mx^2 - 2xx' + 2mnx + 2mxy' + 2ny' + n^2 = r^2 - x'^2 - y'^2$$

$$(1+m)x^2 + x(2mn + 2my' - 2x') + (2ny' + n^2 - r^2 + x'^2 + y'^2) = 0$$

Resolvemos la ecuación que nos queda y obtenemos dos valores de x, uno de los cuales, denotado por x'' sustituye a x en la ecuación de la recta:

$$y = mx'' + n$$

Lo que nos dará el valor de y correspondiente para cada x.

A.3. Punto de corte entre dos circunferencias

A continuación se mostrarán los cálculos referentes a los puntos de corte entre dos circunferencias dadas de la forma $(x-x')^2 + (y-y')^2 = (r')^2$ y $(x-x'')^2 + (y-y'')^2 = (r'')^2$:

En ambas desarrollamos los cuadrados:

$$x^2 + x'^2 - 2xx' + y^2 + y'^2 - 2yy' = (r')^2$$
$$x^2 + x''^2 - 2xx'' + y^2 + y''^2 - 2yy'' = (r'')^2$$

Le restamos la segunda a la primera:

$$x'^2 - 2xx' + y'^2 - 2yy' - (x''^2 - 2xx'' + y''^2 - 2yy'') = (r')^2 - (r'')^2$$

Agrupamos los términos según multipliquen a la x o a la y:

$$2xx'' - 2xx' + 2yy'' - 2yy' - x''^2 + x'^2 - y''^2 + y'^2 + (r'')^2 - (r')^2 = 0$$
$$2x(x'' - x') + 2y(y'' - y') - x''^2 + x'^2 - y''^2 + y'^2 + (r'')^2 - (r')^2 = 0$$
$$2x(x'' - x') = 2y(y' - y'') + x''^2 - x'^2 + y''^2 - y'^2 - (r'')^2 + (r')^2$$
$$x = \frac{y(y' - y'')}{x'' - x'} + \frac{(x''^2 - x'^2 + y''^2 - y'^2 - (r'')^2 + (r')^2)}{2(x'' - x')}$$

Por comodidad al ser todo constantes haremos los siguientes cambios de variable:

$$B = \frac{y' - y''}{x'' - x'}$$
$$A = \frac{x''^2 - x'^2 + y''^2 - y'^2 - (r'')^2 + (r')^2}{2(x'' - x')}$$

Lo cual no dejaría lo siguiente:

$$x = yB + A$$

Sustituimos la x en la ecuación de una de las circunferencias:

$$(yB + A - x')^2 + (y - y')^2 = (r')^2$$

Desarrollamos los cuadrados y agrupamos:

$$(yB + A)^2 + x'^2 - 2(yB + A)x' + y^2 + y'^2 - 2yy' = (r')^2$$
$$(yB)^2 + A^2 + 2yAB + x'^2 - 2(yB + A)x' + y^2 + y'^2 - 2yy' = (r')^2$$
$$(yB)^2 + y^2 + 2yAB - 2(yB + A)x' - 2yy' + A^2 + x'^2 + y'^2 - (r')^2 = 0$$
$$(B^2 + 1)y^2 + 2y2AB - (B + A)x' - y') + (A^2 + x'^2 + y'^2 - (r')^2) = 0$$

Resolvemos la ecuación de segundo grado para obtener dos valores de y, después simplemente sustituimos en la ecuación de la x y obtenemos los puntos que estábamos buscando.

Bibliografía

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. L., editors (2007). *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer.

Osherove, R. (2014). *The Art of Unit Testing*. Springer.