A Domain Specific Language for Constructive Geometry Matematical Models of Technology

Mariona González
Víctor Martín
Jordi Massó
Laura Peña
Ernest Sorinas
Farners Vallespí

1 Introduction

The aim of this project is to develop a small computer language to describe geometric constructions with compass and straightedge. The goals are first to automatically deduce the algebraic equations for the construction points or sets and second, to generate javascript code for interactive visualization of the construction in a standard internet browser.

In order to do all this, we have used Racket and JavaScript. We would like to give a quick introduction to Racket, which is a general-purpose programming language of the List family, meaning that each order is given in between parenthesis, listing a prefix operator which is followed by its arguments. For example, in order to write a = 3 + 2, we would have to write *(define a (+ 3 2))*. This language mainly serves as a platform for language creation, design and implementation and is used in many different contexts such as scripting, computer science and research.

2 Algebraic equations

Using Racket we want to know the algebraic equations for any geometric construction drawn using compass and straightedge. In order to do that, we have written the code to obtain the algebraic equations for five basic constructions. Those are the following ones:

- **point**: given the two coordinates a and b of any point we calculate the algebraic equations using *(eqs-point a b)*.
- **line**: given two points p and q the algebraic equations are given by *(eqs-line p q)*.
- **circle**: given the center p and the radius r the algebraic equations are given by *(eqs-circle p r)*.
- **intersection**: given two objects, *(eqs-intersection obj-1 obj-2)* returns the equations we need to solve to find their intersection.
- **distance**: given two points p and q, *(dist-points p q)* returns the euclidean distance between them.

The one below is an example of how calling a simple construction looks like. In this case, given two points $A = (0, 0)$ and $B = (1, 0)$ we want to know the coordinates of the intersections between the line AB and the circle with center A and radius the distance between A and B.

```
(define A (eqs-point 0 0))
(define B (eqs-point 0 1))
(define a (eqs-line A B))
(define r (dist-points A B))
(define c1 (eqs-circle A r))
(define CD (eqs-intersection a c1))
```

If we define `points` as

```
(define points (CD 'x 'y))
```

then `points` returns us the system of equations we need to solve in order to know the coordinates of the intersections wanted.

2.1 Step construction description

Given any construction we want to know for each element of the set how we constructed it. For doing that, we have a function called *explain* that returns for each element of a set the initial element we have needed to construct the first one. In our example, after running *(define (explain cmds))* where *cmds* is a list of elements defined as:

```
(define example '(
   [A (point 0 0)]
   [B (point 0 1)]
   [a (line A B)]
   [r (len A B)]
   [c1 (circ A r)]
   [CD (cut a c1)]))
```

Then, calling *(explain example)* we obtain:

```
Sigui A el punt de coordenada x = 0 i coordenada y = 0.
Sigui B el punt de coordenada x = 0 i coordenada y = 1.
Sigui a la recta que passa pels punts A i B.
Sigui r la distància entre A i B.
Sigui c1 la circumferència de centre A i radi r.
Sigui CD la intersecció entre a i c1.
```

2.2 Algebraic equations for different programs

Once we have our final construction, Racket give us the equations in the way we have explained in the introduction. Using the example below, what we obtain after we ask Racket to print the equations on screen is the following:

```
'((- a_1 0)
  (- b_1 0)
  (- a_2 0)
  (- b_2 1)
  (- (* (- y b_1) (- a_2 a_1)) (* (- x a_1) (- b_2 b_1)))
  (- a_3 0)
  (- b_3 0)
  (- a_4 0)
  (- b_4 0)
  (- a_5 0)
  (- b_5 1)
  (- (+ (sqr (- a_4 a_5)) (sqr (- b_4 b_5))) (sqr r_1))
  (- (+ (sqr (- x a_3)) (sqr (- y b_3))) (sqr r_1)))
```

So, if we want to solve this system using an outside program, we have to convert Racket language in a way that the program we are going to use can solve them. In order to do that, we made a function called *pretty-print* that turns Racket expressions into normal expressions. This means that *pretty-print* converts (+ 3 2) into $3 + 2$.

Depending on the program we want to use to solve the equations we add some expressions that are needed. We have converted Racket equations in three different ways. Those are the following ones:

2.2.1 Wolfram alpha

Using *(WolframAlpha pols)* we convert our Racket expression in a way Wolfram Alpha can solve the equations. In our example, this means that after running *(WolframAlpha points)* function we have:

```
GroebnerBasis[{a_1-0=0, b_1-0=0, a_2-0=0, b_2-1=0, (y-b_1)*(a_2-a_1)-(x-a_1)*(b_2-b_1)=0,
a_3-0=0, b_3-0=0, a_4-0=0, b_4-0=0, a_5-0=0, b_5-1=0, ((a_4-a_5)^2+(b_4-b_5)^2)-r_1^2=0,
((x-a_3)^2+(y-b_3)^2)-r_1^2=0}, {b_4, b_1, a_5, b_3, a_4, a_3, y, a_1, b_5, a_2, x, b_2, r_1}]
```

2.2.2 Maple

Similar to the previous section, we have a function *(maple-solve pols)* that converts Racket expression in a way Maple can solve it. That means that we print on screen an expression that starts with "Solve" followed by all the equations and the variables from our equations. Also, to know the variables of our equations we have a function called *(vars eqs)* that returns all the variables that are in a list of equations and it has been included in the *maple-solve* procedure. Using our example, we will have:

```
solve({a_1-0=0, b_1-0=0, a_2-0=0, b_2-1=0, (y-b_1)*(a_2-a_1)-(x-a_1)*(b_2-b_1)=0, a_3-0=0,
b_3-0=0, a_4-0=0, b_4-0=0, a_5-0=0, b_5-1=0, ((a_4-a_5)^2+(b_4-b_5)^2)-r_1^2=0,
((x-a_3)^2+(y-b_3)^2)-r_1^2=0}, {b_4, b_1, a_5, b_3, a_4, a_3, y, a_1, b_5, a_2, x, b_2, r_1})
```

In this way, we will be able to copy this text into Maple in order to solve the equations.

2.2.3 JavaScript

In this case, we want to convert Racket expression into a JavaScript expression. For that, we will use *jsexpr* that is a function included in Racket. So, we only have to run what we want and *jsexpr* will return a JavaScript expression. Also, in this case we don't want the complete equations, we just want for each element its parents so, we don't describe each element using equations, just the initial points. The other elements of the set will be define as lines, circles, distances or intersections using the initial points. For example, using the construction we have been using in this section, first of all we have to define it as:

```
(define A (json-point 0 0))
(define B (json-point 0 1))
(define a (json-line A B))
(define c1 (json-circle A B))
(define CD (json-intersection a c1))
(define exemple (list A B a c1 CD))
(jsexpr->string (elements exemple))
```

And, after running *elements exemple* where we have included the *jsexpr* function we have:

{\"5\":{\"parents\":{\"host1\":\"3\",\"host2\":\"4\"},\"type\":\"intersection\",
\"color\":\"#c74440\",\"id\":\"5\",\"negRoot\":false,\"hidden\":false},
\"1\":{\"parents\":{},\"type\":\"point\",\"color\":\"#c74440\",\"x\":0,\"y\":0,
\"id\":\"1\",\"hidden\":false},\"4\":{\"parents\":{\"point2\":\"2\",\"point1\":\"1\"},
\"type\":\"circle\",\"color\":\"#388c46\",\"id\":\"4\",\"hidden\":false},
\"3\":{\"parents\":{\"point2\":\"2\",\"point1\":\"1\"},\"type\":\"line\",
\"color\":\"#2d70b3\",\"id\":\"3\",\"hidden\":false},\"2\":{\"parents\":{},
\"type\":\"point\",\"color\":\"#c74440\",\"x\":0,\"y\":1,\"id\":\"2\",\"hidden\":false}}

3 Visual representation

In order to visualize the desired geometric figures we have developed two codes in different languages, one in Racket and one in JavaScript, that complement each other and are used to generate code and visualize the results in a navigator such as Mozilla Firefox or Google Chrome.

3.1 Racket code

Our racket written program, `conversorAJS.rkt`, basically takes a script stating the steps that need to be followed to construct a geometric figure and creates the specific script needed to develop this construction in JavaScript.

To call the program we have to use the order `(update path)`, where `path` contains the full path to the JavaScript code implementing the functions.

To create the geometric figures we can only use the following instructions:

- `given`: used to declare the points or variables that will be needed from the start
- `line`: used to create a line that goes through two points
- `circ`: used to create a circle that goes through 2 points
- `cut`: used to calculate the intersection between two lines, two circles or a line and a circle
- `first` and `second`: used to get specific elements from a list

This is an example of what would a set of steps look like. In this case, given two starting points A and B, this implements the construction of two equilateral triangles $\triangle ABC$ and $\triangle ABD$.

```
(define example '(
    [given A B]
    [c1 (circ A B)]
    [c2 (circ B A)]
    [CD (cut c1 c2)]
    [C (first CD)]
    [D (second CD)]
    [a (line B C)]
    [b (line C A)]
    [c (line A B)]
))
```

Once given this list, the program would generate the JavaScript code needed to create the initial variables giving them random values and initiate them, the code needed the call the functions that will create each simple figure (such as points, lines and circles) and that will calculate the intersections between them, and finally the code needed to call the functions that will draw the resulting figure in order to visualize the construction.

All this code is then inserted into the file containing the JavaScript code. Each peace of generated code is inserted into a specific place in the file where initially we can find specific comments such as `/*InitialData*/`, which is where we would insert the code to give the initial values to the starting variables substituting this comment.

3.2 JavaScript code

In order to have a visual representation of the geometrical constructions made in Racket, we use a JavaScript code inside an HTML script, which allows, thanks to the usage of g9, to have an interactive front-end in which it is really easy to appreciate how these constructions work.

Our JavaScript code, implemented in `funcions_geometriques.html` and `template.html` (the first file is intended to be modified by the racket function, and the second one is intended to be a copy from which the first file can be reset) is structured as follows: First of all, the declarations of the different types of elemental geometric objects (points, vectors, lines and circles) are given, some of them, in terms of the elements that generates them (for example, the construction of a line depends on the previous definition of two points through which the line will pass). Once we have defined this elements, we can define some basic functions to control how this elements will interact, namely the functions to define intersection of lines, the intersection of circles, and the intersection of a line with a circle.

Now, we can define the functions that, using g9, will draw this elements in our front-end. Finally, we have our main function, `var render`, which we will next explain, and some functions regarding the interaction of the HTML buttons of our front-end with the drawing itself.

The function render is preceded by the comment `/*initialData*/` and by default only has three comments: `/*bloc1*/`, `/*bloc2*/`, `/*bloc3*/`, which will be used by the Racket code to know, respectively, where to insert the declaration the random values that initialize for the initial data (the set of basic elements upon which the geometrical construction is built), to create the code regarding the declaration of these elemental variables, to create the lines that will define the construction of the new elements built using the previous ones, and to generate the code that will call the drawing functions that will be used by the front-end.

Finally, we also have created two examples `triangle.html` and `pentagon.html`, which give an step-by-step construction of the equilateral triangle and pentagon, respectively. It makes use of the buttons `Next Step` and `Reset` in order to advance into the mathematical construction or restart it from the beginning, respectively.

4 A study of g9 library

In many parts of this project we have used a JavaScript library called g9.js. Now, in order to understand precisely what are we working with, we will do a little explanation about it.

4.1 Objective

The point of g9.js is allowing the user to create interactive graphics in JavaScript code with very easy and understandable code. In fact, all that the user has to do is write the render function which displays a picture, and the library will work its magic so that when you drag any point (we'll see later that we can actually restrict the points that the user can drag) the picture automatically changes to one that has the point that the user dragged in the place that he wanted it to be, but keeping the relation between the objects in the picture.

4.2 Working

Using g9 is very easy and intuitive (if the user knows the language, of course). The key parts of the code are:

- Declaring the initial data.
- Displaying some graphics in the render function using this initial data.
- Calling g9, and optionally changing the display settings.

This three parts are compulsory for the code to work but the last one (the g9 call) can always be like this

```
g9(initialData, render)
    .align('center', 'center')
    .insertInto('#FileName')
```

Listing 1: g9 call

and the declaration of initial data is just a list of key-value pairs, something like a table or a hash, and its format is very intuitive, with the example below (Listing 2) one can easily deduce how to do any initial data declaration.

```
var initialData = {
    var1: 10
    var2: 6.1
    ...
    varn: -1.4
}
```

Listing 2: Initial data declaration (Example)

So essentially all the work is done in the render function. There is where we create objects related to this initial data we have and this relations are the ones that have to remain true when we drag the graphics later on.
In this render function one can define variables, work with them, do loops... Anything you need to draw the graphics. To give an example, a very easy render function (which only draws a point in the same position regardless of the initial data) would be like this:

```
function render(data, ctx){
    ctx.point(3, 2)
}
```

Listing 3: Render function (Example)

4.3 Special options

What we have seen so far are the basics, although a lot of complicated graphics can be made only with this. However, there are other options or settings that you can change in the code. For example,

- Choose what objects will be able to be dragged and in what range. By default, each object can be dragged anywhere, but with the function **affects** the user can actually decide to let only some of them move and, for example, one could limit the movement of one object in the X-axis but leave it free in the Y-axis.
- Visual settings: One can change the colour of the objects, the thickness of the lines and all this kind of things related to the visualization of the objects. This is made using keywords like **stroke** or **stroke-width**.
- Inserting images: It is easy to insert images in the graphics, which will be considered objects and so will be (by default) able to be dragged. Obviously one can not move the parts of the image, the image is considered one object itself. The insertion is made by the command **ctx.image** and specifying the path to the image, the position of it in the graphics and how big you want it to be.

```
1     ctx.image('path_to_image_example.jpg', data.x, data.y, 100, 100)
```

- Mathematical functions: One can use a list of mathematical functions like **max** for calculating the maximum/minimum of a set using the command **Math.max**, in this case.

4.4 Inside g9

So the question at this point is clear: How does g9 work its magic? That's what we have been studying most of the time that we dedicated to this library: reading and understanding every part of the functions in the library.
During this process we faced some problems. The first and most obvious one was learning the language, we had never worked with JavaScript before so we had to do an important investment of time learning some basics. Reading and understanding an algorithm created by another programmer with no indications is something that was surprisingly harder than we expected.
After the research we can give the main idea of how the code works: When the user drags any point in the graphic, the function *desire* is called. The point of this function is to minimize the function *cost* related to each of the objects in the graphics, where this function *cost* evaluates something that we could call the 'distance' between the objects in the graphic and the objects in the place where we are trying to move them.
This function *cost* is minimized using the gradient descent method. This is an iterative method similar to Newton's Method. In each step we move in the steepest descent (gradient's) by a small amount. This small amount can be a small constant or, like in our case, it can be calculated each iteration in order to fulfill some error conditions.