An Embedded Geometrical Language in Haskell: Construction, Visualisation, Proof

Maria Grima and Gordon J. Pace

Department of Computer Science, University of Malta

Abstract. Geometric constructions based on compass and straight-edge have been thoroughly studied and explored. In this paper, we present a language embedded in Haskell, to describe, manipulate and analyse such constructions. The use of embedded languages has been explored in various specialised domains, and have been shown to be an excellent front-end to describe such specialised programs, enabling, for instance, the description of families of constructions as functions in the host language, which produce different specialised programs based on input parameters. In particular, we are interested, not only in providing a framework within which one may describe a construction and families of similar constructions in an algorithmic fashion, but also in providing facilities to both test and verify certain properties of constructions, such as equivalence of constructions, or equality of angles and distances in a construction.

1 Introduction

Various tools exist, illustrating the concepts of geometric constructions, allowing the drawing and manipulation of user-given constructions. The main drawback with most interfaces to such systems is that to have a user-friendly front end to the drawing program, one sacrifices desirable features (or at least, easy access to such features), such as reuse of constructions, constructions which may require the repetition of certain actions until a condition is met, and other similar features. In this paper, we present the design of a domain-specific language for geometric constructions, and the use of embedded languages to enable the creation and analysis of constructions in this language. In particular, we are interested not only in providing a framework within which one may describe a construction and families of similar constructions in an algorithmic fashion but also in providing facilities to both test and verify certain properties of constructions, such as equivalence of constructions, or equality of angles and distances in a construction.

1.1 Geometric Constructions

Geometric construction on the plane has proved to be a fascinating mathematical area since Greek antiquity, with the idealised straight line and the circle being considered as basic figures. Since then, mathematicians have been studying these geometric constructions, where one starts with a set of points, lines and edges, and is required to construct new points, lines and circles satisfying a given specification. By constraining the tools used to calculate the new points, lines and shapes, one is essentially setting an axiomatic base, and by investigating the expressiveness and limits of the tools, one is actually studying completeness results of that axiomatic base. Compass and straight-edge constructions limit the tools to a collapsible compass, which can be used to draw a circle centred on a known point, with another known point on the circumference (the compass is said to be collapsible in that it collapses as soon as it is pulled off the plane, essentially stopping direct transfer of distances using the compass), and an unmarked straight-edge, which can be used to draw a line between two known points (it is said to be unmarked in that measurements are not allowed using the straight-edge). Such constructions, sometimes also known as Euclidean constructions, have been explored for centuries, and various interesting results exist, both in terms of constructions which can be achieved (such as bisecting an angle, bisecting a line, and constructing a regular pentagon), and ones which cannot (most famously trisecting an arbitrary angle, squaring the circle and doubling the cube). All such geometric constructions are based on two basic concepts: equidistance established by the use of the compass and collinearity established by the use of the straight edge.

A geometric construction is an algorithm, a step-by-step process, that builds a geometric figure or model. Such constructions are taught in schools to illustrate basic geometric concepts and rudimentary notions of a constructive proofs. Some researchers claim that the use of geometric constructions can sustain proofs and help to make geometric relationships more understandable through their visual representation [San98]. From a pedagogic point of view, they can also provide the motivation required for students to solve problems by reasoning about them [EA02].

1.2 Embedded Languages

When designing algorithms restricted to a particular domain, it may be beneficial to design a language that is targeted to that domain rather than using a general-purpose one. However, the initial effort required to develop such a domain-specific language and the difficulties in evolving the language as changes are requested, are often the cause for failure in complete development of such languages, especially when its use will not be so extensive. Similarly, establishing proper semantics for a new language also requires a great deal of effort. One approach which alleviates some of these problems is that of embedding the domain-specific language inside a general-purpose programming language [Hud96a,Hud98]. Using this approach, one designs a domain-specific library, which enables programs in the domain-specific language to appear as data objects inside the host language. In this manner, one gets to borrow features from the host language automatically, for things such as sub-program definition and control structures. Although the basic components of the domain-specific language are still to be designed, and built in such a way that they do appear as part of a normal program in the host language, borrowing features of the host language drastically reduces language development time. Furthermore, tools such as interpreters and compilers are automatically available, since the domain-specific programs are also programs in the host language. All that one needs to do is to provide an interpretation function (or multiple ones) of the basic features of the domain-specific language. Arguably, one of the major advantages of this approach, is that the host language acts as a meta-language of the domain-specific language, allowing the program to generate regular domain-specific programs, or modify or analyse them accordingly.

It is important that the host language is flexible enough (both in terms of abstraction features and syntactic features) to enable the designer of the domain-specific language to ensure that domain-specific programs are an indistinguishable part of a program which mixes domain-specific and host language features. Haskell [Jon03] has been shown to provide an excellent infrastructure to embed languages, and there have been several successful implementations of domain-specific embedded languages hosted in Haskell addressing various domains, including geometric region analysis [HJ94], animation [EH97], hardware description [BCSS98] and music composition [Hud96b].

In this paper, we look into the design of an embedded domain-specific language to describe compass and straight-edge constructions in Haskell. Geometric constructions will thus appear within Haskell programs, with a library of functions to allow the visualisation of such constructions and their analysis. In particular, we emphasise the analysis aspect, with functions to test and verify properties. Although the aim is primarily to explore the embedding of geometric constructions in a strongly typed functional language, and not building a teaching-aid for geometric constructions, we believe that the use of a tool with such a programming language and access to analysis techniques can be an excellent way to introduce students to the concepts of program testing and verification.

2 Embedding Geometric Constructions

In keeping with the programming style of the host language, we have embedded geometric constructions as functions. The basic constructors of the language clearly are the drawing of lines and circles based on known points, and the identification of points as intersections of known lines and circles. Although constructions are usually given in a sequential manner, the order of the instructions is conceptually not a strict one, in that one should be able to perform any instruction, as long as the inputs are already known. Consider the following typical instructions explaining how to find the midpoint of two given points p_0 and p_1 (as shown in figure 1):

1. Draw a circle c_0 centred on p_0, passing through p_1.
2. Draw another circle c_1 centred on p_1, passing through p_0.
3. Find the intersection points of circles c_0 and c_1, calling them p_2 and p_3.
4. Draw a line l_0, passing through p_0 and p_1.

5. Draw another line l_1, passing through p_2 and p_3.
6. Find the intersection of lines l_0 and l_1. This is the mid-point of p_0 and p_1.

Clearly, the order of steps 1 and 2 is irrelevant, but both must be performed before the third step. Step 4 depends only on the input, and could thus be performed earlier. Although we could have kept the exact order in which the instructions are specified, through basic combinators which specify a strict ordering, possibly through the use of monadic constructs, we prefer the view that, unless there is a dependency, the order of instructions is not important, and thus could be done in any order. For anyone familiar with Haskell, the following description conveys precisely this meaning, and is the spirit of the constructions we sought to create:

```
midpoint (p0, p1) = midpoint
  where
    c0 = circle (p0, p1)
    c1 = circle (p1, p0)

    [p2, p3] = intersections (c0, c1)

    l0 = line (p0, p1)
    l1 = line (p2, p3)

    midpoint = intersection (l0, l1)
```

A straightforward interpretation of the above piece of code would be to consider points as actual coordinates, and lines and circles as pairs of concrete coordinates. The function could thus be used to evaluate the midpoint of actual points on the plane. However, we want to provide other interpretations of the description, and thus a shallow embedding would not suffice our purposes. The construction is a deeply embedded, structural description, which can be evaluated in different ways, for different purposes.

2.1 Strongly-Typed Shapes

The most crucial aspect in embedded languages is probably the type system used for the domain we are dealing with. A sound type system guarantees that a user is eventually restricted to use each function with data of the right type. At the same time, it must ensure that enough flexibility is provided for the user to be able to use the available functions in all required ways. In other words, a domain-specific embedded language must provide a type system that is sound and complete. An approach which involves phantom types [Rhi03] — parameterized types whose instances are independent of the type parameters — is one possible way to meet these objectives. Another alternative would be to use type-classes to enable overloading of certain functions like line `intersection`, but we opt for the former approach to avoid cluttering function types with type constraints.

```
data Shape a = Shape UntypedShape

data UntypedShape =
    UntypedCircle    (UntypedShape, UntypedShape)
  | UntypedLine      (UntypedShape, UntypedShape)
  | UntypedIntersect (UntypedShape, UntypedShape)
  | ...

data Circle = Circle
data Line   = Line
data Point  = Point
```

Since the user would only be able to view the `Shape` parametrised type and the three dummy types `Circle`, `Line` and `Point`, and create instances of shapes through constructor functions we provide, we can enforce type safety through the use of strongly typed constructor functions:

```
circle :: (Shape Point, Shape Point) -> Shape Circle
circle (Shape p0, Shape p1) = Shape (UntypedCircle (p0, p1))

intersection :: (Shape a, Shape b) -> Shape Point
intersection (Shape s1, Shape s2) = Shape (UntypedIntersect (s1, s2))

...
```

2.2 Non-deterministic Constructions

Most descriptions found in textbooks use the reader's visual model to refer to shapes to resolve ambiguity, in particular that induced through the non-determinism found in order of the results of finding intersection points of a circle and another shape. One solution is to use conditional constructs which, for example, enable the user to identify whether two points are equivalent, or whether they lie on the same side of a line. Another solution, is to give a deterministic interpretation to intersection. One such interpretation is the ordering of points on a shape. The intersection points of two shapes will then be given ordered by the ordering on the first shape. We take the ordered point approach, with points on a circle starting on the given point on the circumference and turning clockwise while the points on a line are ordered in the direction of the vector subtended between the two given points. However, we also introduce means of reasoning conditionally about points through equivalence checking to simplify certain constructions.

For example, consider the problem of constructing an equilateral triangle — given two points p_0 and p_1, it is required to identify a third point p_2, such that all three points are equidistant. Clearly, as long as p_0 and p_1 are distinct points, one can find two possible solutions to this problem.

```
equilateral0 :: (Shape Point, Shape Point) -> [Shape Point]
equilateral0 (p0, p1) = ps
```

```
  where
    c0 = circle (p0, p1)
    c1 = circle (p1, p0)

    ps = intersections (c0, c1)
```

Based on this description, we can draw a regular hexagon with a given side, using the selection function `differentFrom`, which filters a given list of points to ones which are different from a particular given point:

```
hexagon0 :: Shape Point -> [Shape Point]
hexagon0 (p0, p1) = [p0, p1, p2, p3, p4, p5]
  where
    (centre:_) = equilateral0 (p0, p1)

    [p2] = equilateral0 (centre, p1) ‘differentFrom‘ p0
    [p3] = equilateral0 (centre, p2) ‘differentFrom‘ p1
    [p4] = equilateral0 (centre, p3) ‘differentFrom‘ p2
    [p5] = equilateral0 (centre, p4) ‘differentFrom‘ p3
```

Note that here we chose an arbitrary centre (from the two possibilities) for the hexagon.

On the other hand, if we constrain the specification with the extra condition that the three points p_0, p_1 and p_2 turn in a clockwise direction, we can give a deterministic solution using implicit ordering:

```
equilateral1 (p0, p1) = p2
  where
    c0 = circle (p0, p1)
    c1 = circle (p1, p0)

    p2 = first (intersections (c0, c1))
```

Note that `first` takes the head of a list of shapes. Using this code, one can then produce a regular hexagon starting from a given edge:

```
hexagon1 :: Shape Point -> [Shape Point]
hexagon1 (p0, p1) = [p0, p1, p2, p3, p4, p5]
  where
    centre = equilateral1 (p0, p1)

    p2 = equilateral1 (centre, p1)
    p3 = equilateral1 (centre, p2)
    p4 = equilateral1 (centre, p3)
    p5 = equilateral1 (centre, p4)
```

Although the order of execution of the constructors is not crucial in a description of a construction, the use of shared expressions clearly is. In the code for drawing hexagons, the centre of the hexagon is computed once and used four times. However, due to referential-transparency, the code given is identical to that with the description of how the centre is drawn replicated. In practice, the data structure constructed for a hexagon is a tree, with multiple copies of the process used to calculate the centre. If we were to compute, for example, how many circles are drawn in the description of a hexagon, the circles used to compute the centre are added on four times, which is clearly undesirable. Various techniques have been presented in the literature to identify shared nodes in a structure. Explicit tagging of nodes in which every shared node in a structure has to be given a name explicitly by the user is one solution [O'D93]. An alternative is to take a monadic approach, using a state monad to compute tags automatically [BCSS98]. The solution we adopt is that of observable sharing [CS99], which introduces non-updateable references breaking referential transparency in a limited way. The abstract datatypes presented earlier are thus all encased within an additional reference type.

2.3 Parametrised Constructions

One advantage of embedding a language, is that the host language automatically acts as a meta-language, enabling us to define functions which can produce a family of constructions depending on inputs which it is given. For example, one can produce a function, which given a natural number n, returns a construction which given two points, returns a point which is $\frac{1}{2^n}$ of the distance from the first to the second point:

```
approach :: Integer -> (Shape Point, Shape Point) -> Shape Point
approach 0 (p0, p1) = p1
approach n (p0, p1) = approach (n-1) (p0, midpoint (p0, p1))
```

Consider the repetition inherent in the example given earlier with the hexagon construction. One may give a general description which, given the number of sides of a regular polygon, its centre, and an edge of the polygon, produces the list of vertices of the polygon. To do this we start by giving a higher-order construction f, which is given a construction from a pair of shapes to a new shape (all of the same type), and a pair of shapes x and y, and returns an infinite list of shapes corresponding to a Fibonacci-like list: $[x, y, f(x, y), f(y, f(x, y)), \ldots]$:

```
repeatConstruction construction (x,y) =
  x: repeatConstruction construction (y, z)
  where
    z = construction (x, y)
```

An n-sided regular polygon can now be described as taking the first n elements of repeatedly finding the next vertex circumscribing the polygon:

```
regularPolygon n (centre, (p0, p1)) = points
  where
    c = circle (centre, p1)
    nextPoint (p, p') = second (intersections (c, circle (p', p)))
    points = take n (repeatConstruction nextPoint (p0, p1))
```

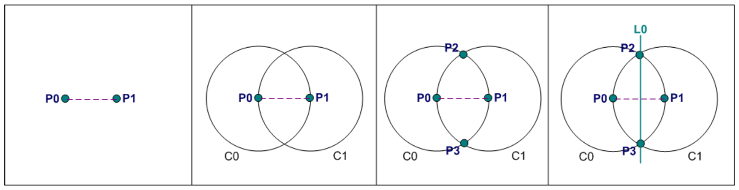

Fig. 1. Construction steps for bisecting segment P0-P1

2.4 Visualisation of Constructions

Since we keep the whole structure of a construction in our datatype, we can use it to transform it into an explanation using textual and visual means. In our geometric construction suite, we provide various visualisation functions, including textual explanations on the lines of the natural language explanation given in section 2, graphical step-by-step explanations in a Postscript document, and an HTML document combining the textual and graphical descriptions. We also enable the user to view constructions in a three dimensional animation by generating input for an external 3D rendering application written for our domain-specific language [Sal07] (see Figure 2).

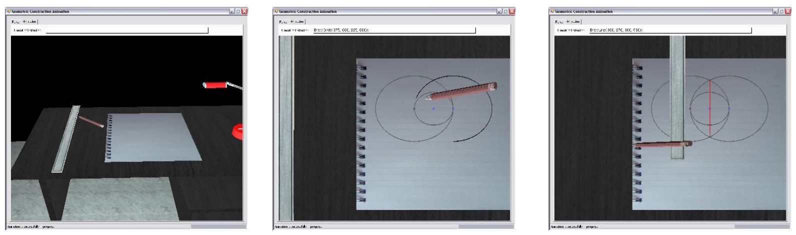

Fig. 2. Selection from the animation sequence for the construction of a perpendicular

3 Analysis of Constructions

Geometric constructions can become rather large and difficult to ascertain their correctness. We provide a number of techniques for the analysis of a given construction, both through the use of testing and theorem proving.

3.1 Testing Constructions

Since we can already evaluate the result of a construction with concrete inputs, testing can be performed simply through the random generation of input points, and checking that the outputs satisfy a given property. We provide two structured approaches to testing constructions:

QuickCheck Testing: As with all Haskell programs, QuickCheck [CH00] can be used to test properties of geometric constructions. To aid the user in testing, we provide a number of functions to calculate information such as distance between points, and angles subtended by three points. Properties can then be written in QuickCheck style in a straightforward manner. For example, to check that the equilateral triangle construction works, we can write the following property:

```
prop_distance (p0,p1) =
  distance (p0,p1) == distance (p0,p2) &&
  distance (p0,p1) == distance (p1,p2)
  where
    p2 = equilateral1 (p0,p1)
```

The function `checkProperty` can then be used to invoke QuickCheck to test the property:

```
> checkProperty prop_distance
OK, passed 100 tests.
```

External Testing: We also provide another means of testing a construction by generating optimised C code to try to falsify the property through random testing. Whereas in QuickCheck, properties are written in plain Haskell, since they are evaluated directly by the Haskell interpreter or compiler, in this second approach, arithmetic and geometric comparison operators which one may need for the property specification language are also deeply embedded in Haskell. The upside of this approach is that expressions can be massaged and optimised before compilation, whereas in QuickCheck, the expressions have to be interpreted in every iteration. Furthermore, since both the constructions and properties are all relative to the inputs, thus guaranteeing that properties are invariant under transformation and scaling of the construction, we optimise further by fixing the first input point to lie at the origin, and the second at $(1, 0)$[1]. This last optimisation improves testing drastically, especially when the construction has a small number of inputs. In the case of the equilateral triangle, it turns out that with a single testing point, the property can in fact be exhaustively verified correct.

3.2 Verifying Constructions

Although testing can be beneficial when trying to find bugs in a construction, one can benefit from the mathematical foundations underlying geometric constructions to actually prove that a construction works as intended. We have implemented an automatic geometry theorem prover, applying the full-angle [Wu87,CG96] method, to enable such reasoning.

[1] Here, we consider the case when the input points are all distinct. The other cases can be checked separately.

The full-angle method enables one to reason about equality of angles, and collinearity of lines. It works by transforming the proof goal into an equation in terms of full-angles, where a full-angle is the rotation required for a particular edge to become parallel to another. The method then proceeds by farming geometric information predicates from the construction, for example deducing that any two points lying on the same circle are equidistant from the centre of the circle. A forward chaining process then follows, in which a number of inference rules are applied to the known predicates, producing more information predicates. Rewrite rules are then used to transform the predicates into full-angle equations to try to prove the original conjecture. The full-angle proof method is not complete on its own, but can be made complete through the use of the full-area method.

Verification proceeds in a very similar manner to testing, in which the user builds a conjecture, where the inputs are taken to denote universally quantified:

```
conjecture (p0, p1) = collinear (p0, p1, p2)
  where
    p2 = midpoint (p0, p1)
```

Applying the `prove` function to the conjecture starts off the proof search, which will result in a proof consisting of the initial predicates (and derived ones), and followed by a proof script of how the result follows in terms of full-angles.

4 Conclusions

The aim of this paper is to discuss the challenges and possible solutions when building a validation and verification based geometrical construction assistant. Our approach was to start with an algorithmic view of the constructions, which can then be visualised at a later stage. The use of embedded languages allowed us to produce regular constructions through a two-staged interpretation of the parametrised constructions. Apart from the actual construction process, we emphasise the analysis of their correctness, enabling straightforward access validation and verification methods.

Various tools exist for the description and manipulation of geometric drawings. For instance, as in our case, Cabri Geometry [PA96] allows drawing of objects on a geometric basis rather than on a perceptual basis. Constructions can also be instantiated and reused as the input to other functions for higher-order constructions. This approach is very similar to the approach that we took to provide the functionality for a user to define constructions as well as being shown the corresponding explanations, but without the capability of analysis. GeoView [BGP04], on the other hand, emphasises verification, and provides constructive capabilities to generate statements representing geometric theorems, and interactive means to build proofs via constructions. The reasoning is all done using the general purpose theorem prover Coq as a back-end. Finally, Geometry Explorer [WF05] also provides access to a full-angle method based prover implemented in

Prolog, but provides only a graphical front-end, thus limiting the complexity of the constructions.

Currently, all parametrised constructions in our system are generated by Haskell programs, using recursion on parameters from the Haskell side of the program. It would be interesting to add stronger condition checks on geometric constructions (such as whether two shapes intersect, or whether two points lie on opposite sides of a line), and thus enable stronger interaction between the Haskell and geometric parts of the code. Given a construction which given two adjacent vertices produces the next vertex of the polygon, one could, for instance, generate the remaining vertices iterating the construction until a vertex is repeated.

We believe that the use of embedded languages, providing us with a meta-language in which to generate, transform, and analyse constructions is a very strong contender for a front-, to middle-end for an educational tool for geometrical constructions. Building a front-end, enabling beginners to draw constructions (which can be automatically translated into programs in our embedded language) would clearly be needed. However, the possibility to describe constructions in a programming environment is too strong to hide away as a possible means of input from the user. Finally, one can connect the language to multiple validation and verification tools, as for example is the case with Lava [BCSS98], which is connected to various model-checkers.

References

[BCSS98] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware Design in Haskell. In *International Conference on Functional Programming 1998*, pages 174–184. ACM SIGPLAN, 1998.

[BGP04] Yves Bertot, Frédérique Guilhot, and Loic Pottier. Visualizing Geometrical Statements with GeoView. *Electronic Notes in Theoretical Computer Science*, 103:49–65, 2004.

[CG96] Shang-Ching Chou and Xiao-Shan Gao. Automated Generation of Readable Proofs with Geometric Invariants II. Theorem proving with full-angles. *Journal of Automated Reasoning*, 17(3):349–370, 1996.

[CH00] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *International Conference on Functional Programming 2000*. ACM SIGPLAN, 2000.

[CS99] Koen Claessen and David Sands. Observable Sharing for Functional Circuit Description. In *Proceedings of the Asian Computer Science Conference 1999*. Springer Verlag, 1999.

[EA02] Pandisico Eric A. Alternative Geometric Constructions: Promoting Mathematical Reasoning. In *Mathematics Teacher 95*, pages 32–36, 2002.

[EH97] Conan Elliott and Paul Hudak. Functional Reactive Animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming 1997*, volume 32, pages 263–273, New York, NY, USA, 1997. ACM Press.

[HJ94] Paul Hudak and Mark P. Jones. Haskell vs. ada vs. C++ vs awk vs . . . an experiment in software prototyping productivity. Technical report, Department of Computer Science, Yale University, 1994.

[Hud96a] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196, 1996.

[Hud96b] Paul Hudak. Haskore music tutorial. In *Advanced Functional Programming, Second International School-Tutorial Text 1996*, pages 38–67, London, UK, 1996. Springer-Verlag.

[Hud98] Paul Hudak. Modular domain specific languages and tools. In P. Devanbu and J. Poulin, editors, *Proceedings of the Fifth International Conference on Software Reuse 1998*, pages 134–142. IEEE Computer Society Press, 1998.

[Jon03] Simon Peyton Jones. *Haskell 98 Language and Libraries: the Revised Report.* Cambridge University Press, 2003.

[O'D93] John T. O'Donnell. Generating netlists from executable circuit specifications. In *Proceedings of the Glasgow Workshop on Functional Programming 1992*, pages 178–194, London, UK, 1993. Springer-Verlag.

[PA96] Dave Pratt and Janet Ainley. The construction of meanings for geometric construction: Two contrasting cases. *International Journal of Computers for Mathematical Learning*, 1(3):293–322, 1996.

[Rhi03] Morten Rhiger. A foundation for embedded languages. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(3):291–315, 2003.

[Sal07] Victor John Saliba. Animated visualisation of geometric constructions. University of Malta, 2007. Introduction to Graphics Assignment.

[San98] Cathleen V. Sanders. Sharing teaching ideas: Geometric constructions: Visualising and understanding geometry. In *Mathematics Teacher 91*, pages 554–556, 1998.

[WF05] Sean Wilson and Jacques D. Fleuriot. Combining dynamic geometry, automated geometry theorem proving and diagrammatic proofs. In *Proceedings of the European Joint Conferences on Theory and Practice of Software (ETAPS) Satellite Workshop on User Interfaces for Theorem Provers (UITP) 2005*, Edinburgh, UK, 2005. Springer.

[Wu87] Wen-Tsun Wu. Basic principles of mechanical theorem proving in elementary geometrics. *Journal of Automated Reasoning*, 2(3):221–252, 1987.