N. Gu, S. Watanabe, H. Erhan, M. Hank Haeusler, W. Huang, R. Sosa (eds.), *Rethinking Comprehensive Design: Speculative Counterculture, Proceedings of the 19th International Conference on Computer-Aided Architectural Design Research in Asia CAADRIA 2014,* 575–584.

IMPROVING GENERATIVE DESIGN BY COMBINING ABSTRACT GEOMETRY AND HIGHER-ORDER PROGRAMMING

ANTÓNIO M. LEITÃO
INESC-ID, Universidade Técnica de Lisboa, Portugal
antonio.menezes.leitao@ist.utl.pt

Abstract. Generative Design (GD) involves the use of algorithms that compute designs. To take advantage of the computational power of computers, these algorithms must be implemented in a programming language. Although most programming languages have the same computational power, they have very different expressive powers. In this paper we focus on exploring the expressive power of languages and we argue that (1) the ability to use abstract geometry as input and (2) the use of higher-order programming dramatically simplifies the implementation of GD algorithms. We illustrate these concepts using a large and complex example that was developed as a case-study.

Keywords. Generative design; abstract geometry; higher-order programs.

1. Introduction

Although modern CAD applications provide a large number of operations for three-dimensional modelling and many pre-defined parameterized designs (e.g., for stairs, windows, and furniture), it remains difficult to manually create and modify complex innovative designs. For these cases, it is preferable to use Generative Design (GD) (McCormack, 2004). GD involves the use of algorithms that compute designs (Terdizis, 2003). To take advantage of the computational power of computers, these algorithms must be implemented in a programming language. The programming language can be textual, visual, or both but, in any case, it must be expressive enough to allow the description of a large range of algorithms.

In spite of the efforts to formally define the expressive power of programming languages (Felleisen, 1991), expressiveness is still intuitively

used to measure how easy it is for a programming language to implement sophisticated ideas. Although most programming languages have the same computational power, they have very different expressive powers.

Our recent research (Leitão et al, 2010; Lopes and Leitão, 2011) has been focused on the implementation of Rosetta, a development environment for GD that takes advantage of common CAD tools, such as AutoCAD or Rhinoceros 3D, while providing the designer with a number of programming languages to choose from, including, among others, Javascript, AutoLISP, and Racket. Compared to other development environments, such as Grasshopper, the main advantage of Rosetta is the emphasis on choice and portability: programs can be written using all supported languages and generate identical models in all supported CAD applications.

In this paper we will focus on exploring the expressive power of these languages and we will argue that programming languages targeted to the GD domain should be able to treat both abstract and concrete geometry as first-class entities, and should allow the definition and use of higher-order functions.

2. Case Study

In order to evaluate our research, we decided to use an actual design as case study, particularly, one that was also sufficiently non-conventional to make it difficult to use common CAD operations. To this end, we implemented a GD program for MVRDV's Market Hall (Boranyak, 2010), a building characterized by an unconventional shape.

One of the advantages of a GD approach is the ability to generate different instances of a design. These instances can then be analysed according to aesthetics and performance criteria, and the results can be feedback into the GD program to generate additional, improved, instances (Kolarevic, 2003). To this end, instead of capturing the exact geometry of the Market Hall, we captured what we consider the underlying ideas of its design, so that we could generate many different instances. Figure 1 shows one such instance.

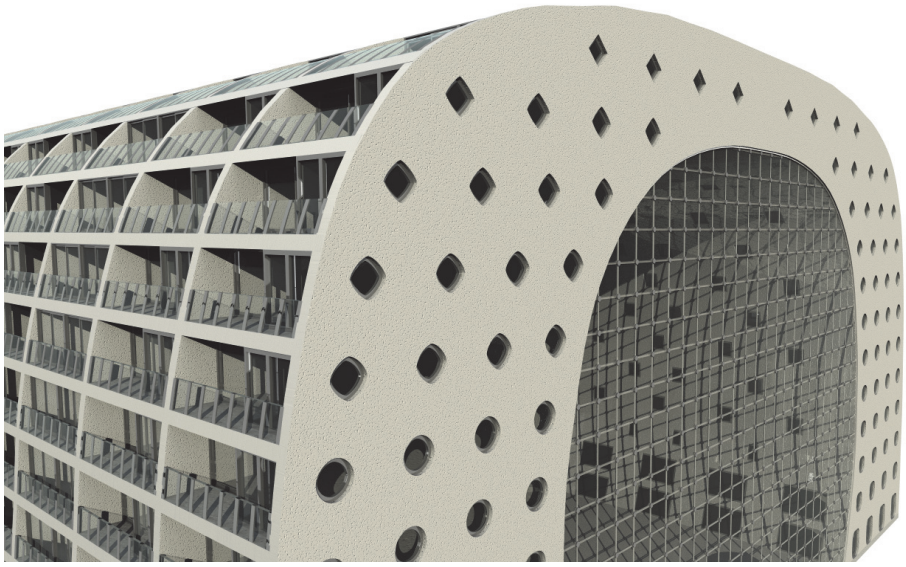

Figure 1. An instance of MVRDV's Market Hall.

The modelling process was divided into four phases: (1) formalization of the building shape, (2) modelling of building elements (walls, slabs, etc.), (3) openings and frames, and finally (4) detailed elements (posts, the elements of a glass facade, etc.). In the first phase, we defined the underlying geometry that captures the overall shape of the building, either as guiding curves and surfaces, or as the mathematical functions that generate them. This geometry was then used as input for the remaining phases of the modelling process.

The decomposition of the process into four phases, each one increasing the detail of the model, creates dependencies between elements. The sequence of phases ensures that the geometry produced in each stage fits in the geometry produced in previous phases. This means that a change in, e.g., the shape of the building, causes a change in the dependent elements, such as window openings, frames, the shape of the walls and slabs, etc.

Besides the phase decomposition, the GD program that was developed explores two important features of Rosetta. The first one is the ability to use abstract geometry as an input to the GD program, allowing a designer to draw curves, surfaces, or solids in a CAD application and use them to derive the geometry of the building and of its elements. The second one is the use of higher-order functions. The following two sections present these concepts.

3. Abstract Geometry and Concrete Designs

In this paper, we use the term abstract geometry to describe any shape representation that, although not present in a concrete design, is used to constraint the generation of that design. In its simplest form, this concept is already available in current CAD tools, for example, in the implementation of shape-forming operations such as sweeping. Here, one abstract shape describing a profile is combined with another abstract shape describing a curve to generate a concrete 3D shape along this curve whose cross-section is the profile.

Our approach takes this intuitive idea and extends it to cover much more complex designs. In fact, in the first phase of our design approach-the formalization of the shape of the building-two curves are used to constraint the shape: one is used to define the longitudinal shape, the other to define the cross-section. However, instead of simply producing the overall shape of the building, these curves are also inspected to identify further geometric characteristics of the building.

As an example, consider the division of the building into longitudinal sections. These sections can be computed in many different ways but two common ones are: (1) by dividing the longitudinal curve into equally spaced sections, or (2) by treating it as a parametrically-specified curve and dividing the parameter domain into equally spaced subdomains. The two approaches produce identical results in the case of a straight line but the results can be very different for buildings with curved longitudinal shapes.

In order to implement this approach, the GD program is parameterized by these abstract curves that guide the generation process. Similarly to the approach of Kilian (2006), this parameterization constraints the final shape: as is illustrated in Figure 2, different instances of the design are generated for different instantiations of these parameters. On the left, a different profile curve was used, while on the right both a different profile and a different longitudinal curve were used.

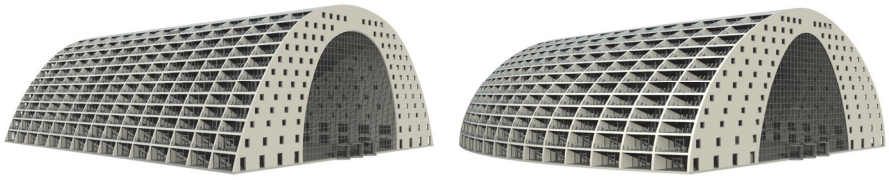

Figure 2. Two different instances of MVRDV's Market Hall.

By providing the designer with a very intuitive metaphor for the overall shape of the building, namely, two simple curves describing the profile and the longitudinal evolution of the building, this approach allows him to easily and intuitively compute different designs that, nevertheless, follow a pre-established style.

This approach is not new and, in fact, has been used in other tools, for example, Generative Components, or Grasshopper. It is its combination with higher-order functions that makes it an even more expressive approach. The next two sections discuss this concept.

4. Higher-Order Functions

A higher-order function is a function that accepts functions as arguments and/or computes functions as results. In spite of the apparent complexity of this concept, it is in fact a very simple one and is widely used in many different areas. The derivative operator, for example, accepts a function as argument, such as $x \to x^2$, and computes another function as result, in this case, $x \to 2x$.

Although ubiquitous in many branches of mathematics, higher-order functions are (unfortunately) frequently avoided in the education of students of Design and Architecture, thus requiring the introduction of additional, unnecessary, concepts. As an example, consider a typical summation operation, e.g.

$$\sum_{i=1}^{10} i^2 = 1 + 4 + 9 + 16 + \cdots + 81 + 100 \tag{1}$$

The summation accepts an expression and the limits of a numeric sequence, and computes the sum of the value of the expression for each element of the sequence. However, in more formal terms, the expression is, in fact, a function, and thus, the summation is a function accepting another function as an argument. This becomes obvious when we study the definition of the summation operation:

$$\sum(f,a,b) = \begin{cases} 0, a > b \\ f(a) + \sum(f, a+1, b), a \le b \end{cases} \tag{2}$$

Note, in the previous definition, that the parameter f is used as a function, in $f(a)$ and it is this fact that makes summation a higher-order function.

Higher-order functions are relevant for generative design because they are a very convenient representation for parameterized geometry. For example, a function that was defined to accept a curve as argument can be easily transformed into a higher-order function that, instead, accepts as argument a function that represents a curve. As will be explained in the next section, this function can, in fact, describe much more complex entities.

It is important to note that not all programming languages support higher-order functions, but many do, including Python, JavaScript, and all Lisp dialects. The use of higher-order functions in the implementation of programs is known as *higher-order programming*.

As an example of higher-order programming, consider the development of a GD program for a building that contains balconies. During the (generative) design process, the designer might be more concerned about the shape of the building than about the shape of the balcony and, as a result, he might decide to implement a non-sophisticated balcony. In formal terms, this means that the formalization of the design will include a parameterized definition for the building that depends on the (global) parameterized definition of the balcony, as well as on the definitions of other parts such as walls and slabs, i.e.,

$$\text{building}(\ldots) = \ldots\text{wall}(\ldots) \ldots \text{slab}(\ldots) \ldots \text{balcony}(\ldots) \quad (3)$$

$$\text{balcony}(\ldots) = \ldots \quad (4)$$

Later on, the designer might want to experiment several different designs for the balcony. If there is a common theme in all those designs, for instance, when the balcony follows a sinusoidal curve, it might be possible to include the sinusoidal parameterization in the building parameterization, thus allowing an infinite number of different designs based on different values for the amplitude, frequency, and phase of the sinusoid. However, if the sinusoid is just one of many possible designs for the balcony, it becomes impossible to have a single dependency between the building definition and the balcony definition. In that case, the best solution is to transform the balcony into a parameter of the building definition:

$$\text{building}(\ldots, \text{balcony}, \ldots) = \ldots\text{wall}(\ldots) \ldots \text{slab}(\ldots) \ldots \text{balcony}(\ldots) \quad (5)$$

The designer might then design different kinds of balconies, such as the ones represented in Figure 3, and implement the corresponding functions:

$$\text{linearBalcony}(\ldots) = \ldots \quad (6)$$

$$\text{sinusoidalBalcony}(\ldots) = \ldots \quad (7)$$

$$\text{clampedBalcony}(\ldots) = \ldots \quad (8)$$

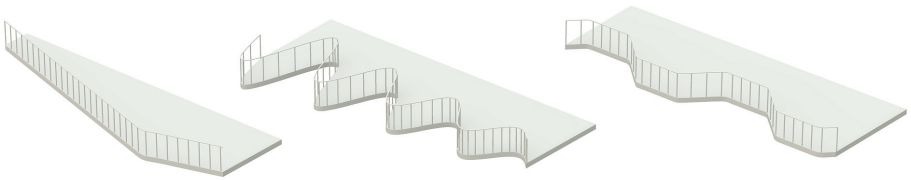

Figure 3. Three different balconies generated by different function provided as argument to the function that creates the building.

Any of these functions can then be provided as an argument to the function that generates the building, as is illustrated in the following function call:

building(…, clampedBalcony, …) (9)

It should be now clear that, besides balconies, many other aspects of a building can be transformed into (functional) parameters of a higher-order function.

It was precisely this that we did for our case study, the Market Hall building. In fact, many of the features of the Market Hall, such as windows and balconies, are implemented as independent functions and the main GD function and many of the functions used in its implementation are higher-order functions parameterized for these elements. Figure 4 shows the same building but where different functions were used for the elements of the balconies.

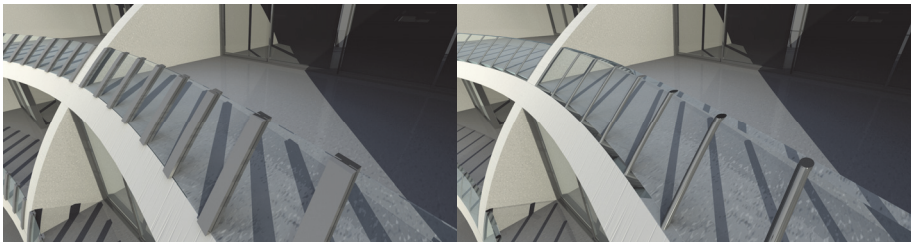

Figure 4. Balconies for the Market Hall generated from the same higher-order function but using different functions as arguments.

5. Abstract Geometry and Higher-Order functions

We discussed, in section 2, how abstract geometry can be an effective means for describing the shape of a building and/or of its elements. In section 4 we discussed the concept of higher-order functions and we exemplified its use. We now discuss the combination of these two powerful concepts for the quick construction and experimentation of large GD programs.

The important premise is that functions can describe geometry. This was the wonderful discovery of René Descartes and it allows us to express complex designs using the language of mathematics.

In fact, by using functions describing curves, surfaces, and solids, we can combine abstract geometries with higher-order functions to provide additional expressive power. This can already be seen in Figure 5, where the abstract longitudinal shape of the building is described, not by a freeform curve, but by a sinusoidal function.

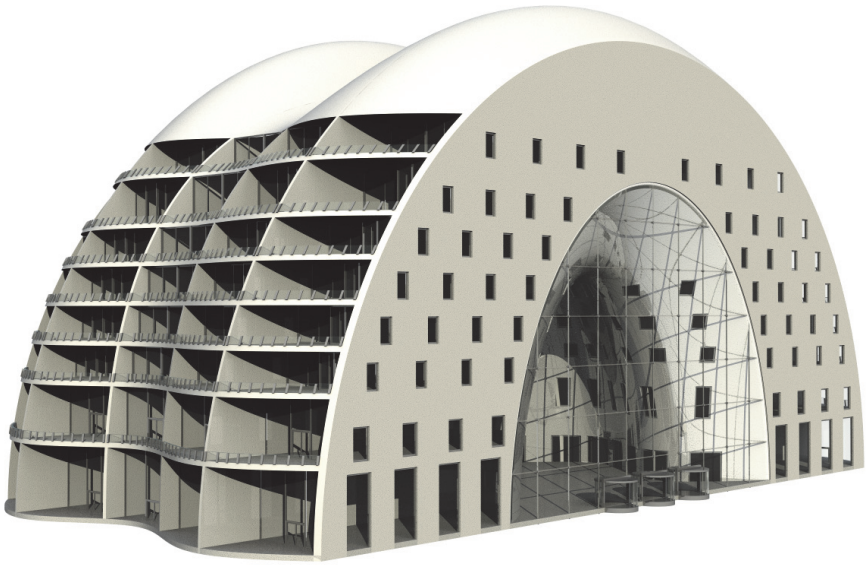

Figure 5. A different instance of MVRDV's Market Hall. The longitudinal shape was defined by a sinusoidal function provided as argument to the function that creates the entire building.

In order to combine higher-order functions with abstract geometry we need a way to convert between these two representations. This problem can be divided into two different sub-problems: converting from functions to geometry and vice-versa.

For the first sub-problem, we use the concept of *sampling*: each (parametric) function that needs to be transformed into some abstract (or concrete) geometry is sampled for different values of its domain. For each sample the function is applied and the resulting geometric coordinate is collected. The resulting set of coordinates is then interpolated to generate a curve, a surface, or solid. The sampling process can be controlled by the user, allowing the specification of the function domain and the sampling rate, or can be automatic, in which case we use an adaptive sampling rate over the function domain.

For the second sub-problem, namely, converting from abstract geometries to function definitions, there are several possible approaches based on any of the curve-fitting or surface-fitting methods available. However, in our current experience, this is a much less needed operation.

6. Conclusions

Higher-order functions are not a new concept and, in fact, they have been used in many different areas, including genetic programming (Binard and Felty, 2008), constructive solid geometry (Davy and Dew, 1995), and shape grammars (Lewis et al, 2004).

In this paper we argue that the combination between abstract geometry and higher-order functions provides a powerful approach to Generative Design. In order to take advantage of the approach, the generative design program must implement functions that accept abstract geometry as arguments but also higher-order functions that accept other functions as arguments. These functional arguments are then used to generate abstract and/or concrete geometry. From the point of view of the programming language, these higher-order functions are just programs that accepts other programs as arguments, thus making the proposed approach an example of higher-order programming.

The combination between abstract geometry and higher-order functions relies on the analytic geometry concepts invented many years ago by René Descartes and on sampling and fitting methods used for the manual or automatic conversion from function definitions to abstract geometry and vice-versa.

We illustrated these concepts on an actual design: the Market Hall building. This a large and complex design that is an excellent test bed for our approach. We found that many of the design features present in this design became much simpler once we implemented them using higher-order programming.

In the near future, we plan to continue to explore the approach by applying it to even larger designs. We are particularly interested in identifying programming language limitations to the use of higher-order programming and in extracting guiding principles that further improve the application of the approach to complex designs.

Acknowledgments

This work was partially supported by Portuguese national funds through FCT under contract Pest-OE/EEI/LA0021/2013 and by the Rosetta project under contract PTDC/ATP-AQI/5224/2012.

References

Binard, F. and Felty, A.: 2008, Genetic programming with polymorphic types and higher-order functions, *in* Maarten Keijzer (Ed.), *Proceedings of the 10th annual conference on Genetic and evolutionary computation* (GECCO '08), ACM, New York, NY, USA, 1187-1194.

Boranyak, S.: 2010, Archetype. Civil Engineering, ASCE, 80(2), 76-79.

Davy, J. and Dew P.: 1995, A polymorphic library for constructive solid geometry, *Journal of Functional Programming,* 5, 415-442.

Leitão, A.; Cabecinhas, F., and Martins, S.: 2010. Revisiting the Architecture Curriculum: The programming perspective, *28th eCAADe*, ETH Zurich, Switzerland, 81-88.

Lewis, J.; Rosenholtz, R.; Fong, N.; and Neumann, U.: 2004, VisualIDs: automatic distinctive icons for desktop interfaces, *in* Joe Marks (Ed.), *ACM SIGGRAPH 2004,* ACM, New York, USA, 416-423.

Lopes, J. and Leitão, A.,: 2011. Portable Generative Design for CAD Applications, *ACADIA 2011*, Banff, Alberta, Canada, 196-203.

Felleisen, M.: 1991. On the expressive power of programming languages, *in* M. Sintzoff and N. D. Jones (Eds.), *Selected papers from the symposium on 3rd European symposium on programming* (ESOP '90),. Elsevier North-Holland, Inc., Amsterdam, The Netherlands, 35-75.

Kalay, Y.: 2004, Architecture's New Media: Principles, Theories, and Methods of Computer-Aided Design. Cambridge, Massachusetts: The MIT Press.

Kilian, A.: 2006, Design innovation through constraint modeling. *International Journal of Architectural Computing*, **1**(4), 87-105.

Kolarevic, B.: 2003, Computing the Performative in Architecture, *21th eCAADe*, Graz, Austria, 457-463.

McCormack, J., Dorin, A., and Innocent, T.: 2004, Generative design: a paradigm for design research. *Proceedings of Futureground*, Design Research Society, Melbourne.

Terdizis, K.: 2003, Expressive Form: A Conceptual Approach to Computational Design. London and New York. Spon Press.