On the Expressive Power of Programming Languages for Generative Design

The Case of Higher-Order Functions

António Leitão[1], Sara Proença[2]
[1,2]INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
[1,2]{antonio.menezes.leitao|sara.proenca}@ist.utl.pt

The expressive power of a language measures the breadth of ideas that can be described in that language and is strongly dependent on the constructs provided by the language. In the programming language area, one of the constructs that increases the expressive power is the concept of higher-order function (HOF). A HOF is a function that accepts functions as arguments and/or returns functions as results. HOF can drastically simplify the programming activity, reducing the development effort, and allowing for more adaptable programs. In this paper we explain the concept of HOFs and its use for Generative Design. We then compare the support for HOF in the most used programming languages in the GD field and we discuss the pedagogy of HOFs.

Keywords: *Generative Design, Higher-Order Functions, Programming Languages*

INTRODUCTION

Generative Design (GD) involves the use of algorithms that compute designs (McCormack 2004, Terdizis 2003). In order to take advantage of computers, these algorithms must be implemented in a programming language. There are two important concepts concerning programming languages: computational power, which measures the complexity of the problems that can be described using the language, and expressive power, which measures the breadth of ideas that can be described using the language. The expressive power is directly related to the human effort needed to describe those ideas in a given programming language.

It is a known fact that any non-trivial programming language is Turing-complete, meaning that almost all programming languages have the same computational power. In what regards their expressive power, however, they can be very different. A given 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.

Expressiveness is intuitively used to measure how easy it is for a programming language to implement complex ideas. There are studies (Felleisen, 1991) that showed that it is possible to rank languages according to a formal definition of expressive power that captures the intuitive meaning of the term. In this ranking, a language that supports,

for example, user-defined functions, (e.g., FORTRAN), is more expressive than one that does not supports them (e.g., BASIC). This does not mean that there is a problem solvable in FORTRAN that cannot be solved in BASIC. What it does mean is that there are problems that require larger implementation efforts from a BASIC programmer than from a FORTRAN programmer.

In this paper, we claim that higher-order functions (HOFs) are an important and powerful programming language feature for GD. HOFs increase the expressiveness of a language, reducing the programmer's effort and improving code reusability.

In the next sections we describe the expressive power allowed by HOFs, we illustrate the advantages of its use in GD, and we analyse the level of support provided by the programming languages currently being used for GD. Finally, we describe a pedagogic approach for teaching HOFs in the realm of GD problems.

HIGHER-ORDER FUNCTIONS

A HOF is a function that accepts functions as arguments and/or computes functions as results. As an example, consider the derivative operator $D(f)$, that accepts a function as argument, such as $x \rightarrow x^2 + 3x$, and computes another function as result, in this case, $x \rightarrow 2x + 3$. This fact can be written as $D(x \rightarrow x^2 + 3x) = x \rightarrow 2x + 3$. Note that the functions that we used as argument and result of the derivative operator did not have a name. These unnamed function are known as *anonymous functions* and there is an alternative notation for them: $\lambda x.2x + 3$. This notation was proposed by the λ-Calculus (Barendregt 1984) and, as we will see, it is adopted by several programming languages. Note, however, that from the point of view of the derivative operator, there is no difference between anonymous functions and named ones. For example, it is well-known that the derivative of the sine function is the cosine function, a fact that we can express using named functions, namely: $D(\sin) = \cos$.

For a different example, consider a typical summation:

$$\sum_{i=1}^{10} i^2 = 1 + 4 + 9 + 16 + \ldots + 81 + 100 \quad (1)$$

Although rarely presented as such, summation is a HOF: it accepts a function (in the previous example, disguised as the expression i^2) and the limits of a numeric sequence (in the example, 1 and 10), and computes the sum of the applications of the function to all the elements of the sequence. Using the appropriate mathematical notation, the example becomes $\sum_{i=1}^{10}(i \rightarrow i^2)$, where it is now obvious that the summation function accepts another function as argument. This is also visible in the formal definition of the summation operation:

$$\sum_{a}^{b} f = \begin{cases} 0 & a > b \\ f(a) + \sum_{a+1}^{b} f & a \leq b \end{cases} \quad (2)$$

Note, in the above definition, that the parameter f is used as a function, in $f(a)$.

Although ubiquitous in many branches of mathematics, HOFs are unjustly considered a complex topic that, as a result, is frequently not taught properly and is not fully supported in a large number of programming languages. However, as we will show, HOFs are an important tool for GD: they dramatically simplify scripting (McCullough, 2006) and they are a very convenient representation for parameterized geometry (Park and Holt, 2010). Programming using HOFs is known as *higher-order programming*.

As an example, consider a balcony in a building. During the (generative) design process, a 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 simplified balcony. This means that the formalization of the design in a programming language includes a parametrized definition for the building that depends on the parametrized definition of the balcony, as follows:

```
balcony(...) = ...
```

```
building(...) =
  ...
  wall(...)
  ...
  slab(...)
  ...
  balcony(...)
  ...
```

Later, when the designer turns his attention to the balcony, he might want to experiment several different designs. In less expressive languages, his only option is to redefine the balcony definition for each experiment. In more expressive languages that support HOFs, a better solution is to transform the balcony into a parameter of the building definition, which then becomes a HOF, as follows:

```
building(...,balcony,...) =
  ...
  wall(...)
  ...
  slab(...)
  ...
  balcony(...)
  ...
```

It is now possible to simultaneously define different balcony functions expressing different balcony designs and treat them as *plug-ins* for the function that represents the building.

Although it might seem that there is little difference between these two approaches, only the second approach allows the designer to easily create a building containing different balcony designs. Moreover, both the function that represents the building as well as the functions that represent the balconies are now less dependent from each other and can be reused in different contexts.

In this paper, we claim that, besides balconies, many other aspects of a design can be transformed into (functional) parameters of HOFs. We illustrate our claim by describing the design of a complex building, more specifically, the Market Hall (Boranyak 2010), represented by functions that accept, as arguments, functions describing the overall shape of the building, functions for the different elements (posts, facade, etc.), functions describing sequences of elements, etc. This representation allows us to generate not only a building that is identical to the original concept, but also an infinite number of variations, expressing different choices for all the function parameters, including those that represent other functions.

HIGHER-ORDER FUNCTIONS IN GENERATIVE DESIGN

In spite of the simplicity of the idea of HOFs, a (generative) designer still needs to think about the best strategy for its use, which is strongly dependent on the current design goals. Returning to the previous example, it is clear that it is not enough to say that the function that creates the balcony is a functional parameter of the function that creates the building; we also need to specify the communication protocol between both functions, namely, which information is expected by the balcony function and which information does it return to the building function. Depending on the programming language being used, the designer might decide to provide part or the entirety of the expected information as *arguments* to the function. Additional information might be provided in global variables or using other language-specific mechanisms. For illustration purposes, we will consider that the balcony function expects the spatial location of a corner of the balcony and the dimensions of the intended balcony. We will assume that the building function provides this information each time it calls the balcony function.

Regarding the design of the balcony, the designer might want to experiment different mathematical functions for its shape. This means that it should be possible to define another (higher-order) function that accepts the shape function and that computes a specialized balcony function that follows that shape. We will now assume the existence of this function and we will only consider the different shape functions that the designer might want to experiment.

One possibility is the use of a sinusoidal shape. To this end, the designer might define a function-

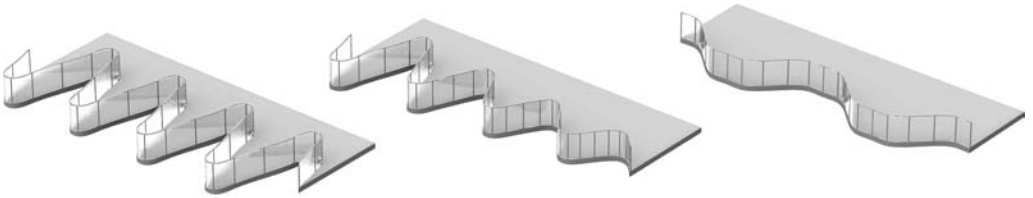

Figure 1
Balconies generated using higher-order functions. From left to right, a sinusoidal function, the product of decaying and sinusoidal functions, and the composition of a clamping and sinusoidal function.

producing function (again, a HOF), that accepts the amplitude, frequency, and phase of the sinusoidal curve:

$$\text{sinusoidal}(\alpha, \omega, \phi) = x \rightarrow \alpha \sin(\omega x + \phi) \quad (3)$$

The balcony produced using the previous function is visible in figure 1, on the left. So far, this is not very different from what he could achieve using only first-order functions. In fact, the advantages of higher-order functions only become clear when we experiment with different functions. For example, an exponential decay describes a process whose output decreases at a rate proportional to its current value, a fact that can be modeled by the following function:

$$\text{decay}(\beta) = x \rightarrow e^{-\beta} \quad (4)$$

The designer can now *multiply* the decay function with the sinusoidal function to produce a function that describes a decaying sinusoid. To this end, he needs to define the *product* $\otimes$ of functions, as an HOF that, given two functions f and g, computes a third function that is $f \otimes g$:

$$\otimes(f, g) = x \rightarrow f(x) \times g(x) \quad (5)$$

As a concrete example, the formula

$$\otimes(\text{decay}(0.1), \text{sinusoid}(-2, 2, 0)) \quad (6)$$

describes the shape of the balcony presented in the center of figure 1.

Many other higher-order operators can be similarly defined. For example, the composition of functions $f \circ g$ can be formally defined as:

$$\circ(f, g) = x \rightarrow f(g(x)) \quad (7)$$

Another example is the *clamping* function, that limits its input to a given range of values:

$$\text{clamp}(\text{inf}, \text{sup}) = x \rightarrow \min(\max(x, \text{inf}), \text{sup}) \quad (8)$$

Using these HOFs, the designer can experiment other kinds of balconies, like the one represented in figure 1, on the right, that corresponds to the clamped sinusoid $\circ(\text{clamp}(-0.7, 0.7), \text{sinusoid}(1, 1, \pi/2))$.

This previous example illustrates the use of HOFs for a very simple case. The concepts, however, scale to much bigger cases, as can be seen in figure 2, which shows a 3D model of the Market Hall building that was fully generated using programming, particularly, HOFs. The complete program is implemented by 332 functions, including 32 HOFs that are used in 143 different places of the program. An analysis of these HOFs reveals that these are used not only to describe an abstract building, in which many of the building elements are functional parameters, but also to implement the *sampling* of parametric functions, i.e., to compute the values of parametric functions for different values of their domain, and to implement *mappings* of functions over collections, i.e., the application of a function to each element of a collection, producing a collection of results.

It is important to note that the generation of the Market Hall 3D model depends on several HOFs, each one implementing a particular part of the design. There are functions that implement the overal shape,

functions that implement balconies, functions that implement elements of balconies, etc. Arbitrary combinations of these functions can then be created to implement different variations of the main design.

Figure 3 shows the "same" Market Hall building, in the sense that it is the result of the same HOF that was used to produce the model in figure 2, but where we provided different functions as arguments, in particular, to describe the overall shape of the building, thus producing a significant variation. While the original building has a shape whose longitudinal evolution is described by a linear function, the variation illustrated in figure 3 shows, among other differences, the use of a sinusoidal function to describe this evolution.

Figure 2
The Market Hall building, entirely generated using programming.

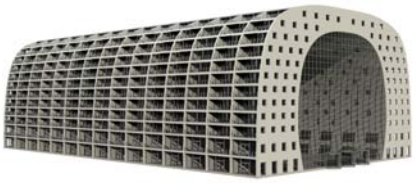

Figure 3
A different instance of the Market Hall building.

HIGHER-ORDER FUNCTIONS IN PROGRAMMING LANGUAGES FOR GENERATIVE DESIGN

Given the usefulness of HOFs, it is not surprising to see them supported in a large number of modern programming languages. There are, however, different levels of support among the different programming languages and even among different versions of the same language. The Java programming language, for example, was released in 1995 but its implementation of anonymous functions appeared only in version 8, released almost 20 years later. The Python programming language did not provide HOFs in its initial implementation, in the late 80's, but the first official release, in 1994, included anonymous functions and some list-processing HOFs. Haskell provides considerable support for HOFs, including automatic partial application but forces the user to work in the less conventional lazy-evaluation paradigm (Hughes, 1989). Finally, there are languages, such as the original BASIC language, that do not event support HOFs.

In this paper, we are particularly concerned with the support for HOFs that is provided by programming languages used in the GD field. The languages that we will discuss are Python, AutoLisp, VBScript, and Grasshopper, but we will also comment on some additional languages used for GD, such as GDL, MAXScript, Processing, PLASM, and Racket.

Python

Python is a programming language that is currently enjoying a considerable momentum in the GD community, particularly, due to fact that it is one of the scripting languages of Rhinoceros 3D. Python provides some pre-defined higher-order functions and also allows user-defined ones. As an example, the following function implements the exact same summation function that was described by formula 2:

```
def summation(f, a, b):
  if a > b:
    return 0
  else:
    return f(a) + summation(f, a+1, b)
```

Similarly, in Python it is possible to provide anonymous functions as arguments, using *lambda* expressions. For example, the expression $\sum_{i=1}^{10}(i \rightarrow i^2)$ becomes:

```
summation(lambda x: x**2, 1, 10)
```

The summation function is an example of an HOF that *accepts* a function as an argument. The composition

of functions is more interesting, as it also *returns* a function as result. Its definition, in Python, becomes:

```
def compose(f, g):
  return lambda x: f(g(x))
```

There is one restriction regarding anonymous functions in Python: they can only contain one expression, which means that it is not possible to include statements in the body of the anonymous function. In practice, this is not a serious limitation, as it is possible to locally define a named function and use it as if it was anonymous. Moreover, many of the programming patterns related to the use of anonymous functions, such as mapping, filtering, and reducing lists can be replaced by *list comprehensions*.

In practice, this means that Python provides good support for programming with HOFs and, thus, can be easily used to implement the GD example that we described.

AutoLISP

AutoLISP is one of the scripting languages of AutoCAD. AutoLISP is a member of the LISP family of languages (and, thus, provides HOFs) and a widely used language for GD, with a huge amount of scripts available on the internet. A cursory look at some of those scripts show that authors were acquainted with some of AutoLISP pre-defined HOFs, particularly, the mapcar function.

AutoLISP also allows user-defined HOFs. For example, the summation function becomes:

```
(defun summation (f a b)
  (if (> a b)
    0
    (+ (f a)
       (summation f (1+ a) b))))
```

Unfortunately, AutoCAD is a dynamically scoped language, and this means that user-defined higher-order functions can cause hard-to-debug problems, particularly, the infamous *downward funarg* and *upward funarg* problems (Moses, 1970). The downward funarg problem occurs when a function provided as argument has a free variable that is shadowed by some variable of the HOF. As an example, consider a function that compute a sum of powers b from 1 to n. Its definition in AutoLISP might be:

```
(defun sum-of-powers (n b)
  (summation (lambda (x) (expt x b))
  1 n))
```

Unfortunately, if we try any computation involving the previous function, we will discover that it does not compute the correct result, because the exponent b used in the anonymous function is *shadowed* by the upper limit b of the HOF. This is a serious problem that can be mitigated but never entirely solved using some name-mangling techniques that prevent name collisions.

The second problem, the upward funarg, is more serious and much more difficult to solve. It occurs when a HOF returns a function as a result, as it happens with the function $\circ(f, g)$. Its definition, in AutoLISP, would look like:

```
(defun compose (f g)
  (lambda (x)
    (f (g x))))
```

Unfortunately, when the returned function has free variables that were bound by the HOF, these variables will lose their current value, thus making the returned function useless. This means that the names f and g that are used in the anonymous function returned by the function compose will not be correctly bound and will cause errors. This is a serious limitation of AutoLISP that restricts the expressive power of the language.

Visual Basic

Visual Basic can be considered a family of languages. There are two main dialects that are used as scripting languages: VBA (Visual Basic for Applications) and VBScript. VBA is a restricted version of Visual Basic that is used, for example, in AutoCAD. VBScript is an even more restricted version that is used in Rhinoceros 3D, under the name RhinoScript. Most of the restrictions are related to the execution environment and are not relevant to our analysis but there

are important restrictions that directly affect the ability to define and use HOFs. One is that VBA does not consider functions as first-class entities, and, as a result, they cannot be passed around just like other values, such as numbers and strings. However, VBA provides a way for calling functions whose name is described by a string. This is not possible in VBScript but, starting from version 5.0, it is possible to do dynamic code evaluation using the functions Eval (for evaluating expressions) and Execute (for executing statements). In both cases, the code to evaluate must be contained in a *string*.

To illustrate the definition of a pseudo-HOF, the following VBScript function mimics the mathematical definition of the $\sum$ function:

```
Function Summation(expr, a, b)
    If a > b Then
        Summation = 0
    Else
        Summation = Eval(expr) + _
          Summation(expr, a + 1, b)
    End If
End Function
```

Differently from the previous languages, where functions were provided as arguments, in the case of VBScript, the function requires a string describe the expression to evaluate:

```
summation("a*a", 1, 10)
```

Note also that the expression has to explicitly refer the variable of the Summation function that contains the correct value to use, which is a serious violation of the encapsulation principle. Moreover, by representing functions with strings, it becomes very difficult to combine functions as this requires the construction of a string that, according to the syntax and semantics of the programming language, represents the intended combination of the expressions. In the general case, this implies writing a compiler. In practice, both the VBA and the VBScript dialects cannot be considered adequate languages for the definition and use of higher-order functions.

Grasshopper

All the languages described previously belong to the category of *textual* programming languages, meaning that programs are written using a one-dimensional sequence of characters. On the other hand, in *visual* programming languages (VPLs), programs are "written" in a bi-dimensional representation consisting of iconic components that can be interactively manipulated by the user. Grasshopper is one of the most popular VPLs for GD and some of its components can be used for higher-order programming.

Figure 4 shows a Grasshopper program using the Evaluate component, that accepts a formula described as a string and a value or sequence of values, and computes the evaluation of the formula for each value received. By changing the string that represents the formula it becomes possible to compute different behaviors, just like was done, e.g., in the VBScript example.

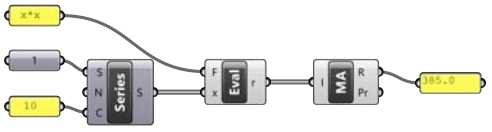

Figure 4
A Grasshopper program that computes the sum of squares of integers from 1 to 10.

Recent Grasshopper versions can even abstract a set of components into a *cluster*, allowing the definition of "functions". Figure 5 shows the same Grasshopper program that is presented in figure 4 but where a cluster was used to abstract the definition of the summation function.

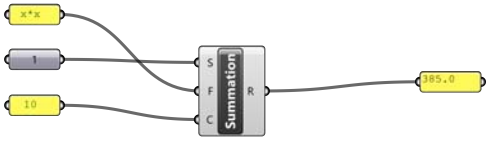

Figure 5
A Grasshopper component that abstracts the summation function.

Unfortunately, by using strings as representations of formulas, we end up finding the same limitations that were discussed for VBScript, making it very cumber-

some to use expressions with free variables or to implement HOFs that return functions as results.

Additional Languages

The previous sections analysed the most used programming languages in the GD field. However, there are several other, less used, languages that could have been included. In this section we briefly review some of them.

GDL. GDL (Watson, 2009) is the scripting language of ArchiCAD, a popular CAD application. GDL is a descendant of BASIC but contrary to its siblings, such as, VisualBasic, GDL did not evolve and has many shortcommings. In particular, subroutines cannot define local variables, receive parameters, or return values, thus making the language completely unsuitable for higher-order programming.

MAXScript. MAXScript (Autodesk, 2006) is the scripting language of Autodesk's 3dsMax. In MAXScript, functions are first-class values and can be provided as arguments or returned as values to/from other functions. As a result, it is trivial to define HOFs and the standard library already implements some. Given that GD scripts tend to have frequent use of mapping operations over collections of values, MAXScript allows the automatic definition of mapped versions of normal functions. In spite of using lexical scope, thus solving the downward funarg problem, each scope is discarded as soon as the execution flow leaves the scope, thus suffering from the upward funarg problem.

Processing. Processing (Reas & Fry, 2010) is a popular programming language specialized for the production of images and animations. The language is a simplified version of Java that suffers from the same limitations of Java for functional programming, namely forcing the use of some basic design patterns, such as the Command, for providing first-class status to functions, and the verbosity required for the definition of even for the most simple higher-order operations.

PLASM. PLaSM (Paoluzzi & Sansoni, 1992) is a functional programming language created for GD with a strong emphasis in higher-order programming, providing many pre-defined operators for function combination. PLaSM uses a rather dense mathematical notation which can be hard to grasp. Recently, a Python front-end for PLaSM was developed, allowing the use of PLaSM operators in a more familiar setting. In spite of allowing the visualization of the generated designs using some well-known standards, such as VRML or SVG, it cannot be directly used with CAD applications.

Racket. Racket (Tobin-Hochstadt 2011) is a modern member of the LISP family of languages, designed for pedagogical and practical purposes. Racket is also one of the languages supported by Rosetta (Lopes 2011), a programming environment providing multiple programming languages, such as Javascript and AutoLISP, and multiple CAD applications, such as AutoCAD and Rhinoceros 3D. Racket provides strong support for higher-order programming, with many pre-defined HOFs and unrestricted user-defined HOFs.

TEACHING HIGHER-ORDER FUNCTIONS

In the last several years we have been teaching a one-semester computer science course for students of architecture that do not have any prior programming experience. Given the increased expressive power provided by HOFs, it was our goal from the beginning to explore the topic during the course. In order to present the concept using ideas that the students can quickly grasp, we explore the summation function as a motivating example, but always using named functions as arguments. When we feel that students have a good understanding of the concept, we explain anonymous functions as simplified functions that are created just for an ephemerous use and then we explain the relation between named and unnamed functions. In order to keep the students interested in the topic we also solve simple architectonic problems, such as designing buildings with shapes defined by functions or computing curves and surfaces from their parametric definitions.

In our experience, students can easily under-

stand the concept of HOF. There is, however, a serious problem in the use of those concepts in programming languages that were not designed to support them. This was the case in the first years, when we had to teach computer science using AutoLISP (Leitão 2010). AutoLISP is a language that is easy to learn but, as we explained before, it is severely limited in its support for HOFs, both downward (i.e., functions that accept functions as arguments) and upward ones (i.e., functions that return functions as results). This forced us not only to avoid talking about upward HOFs but also to waste some time explaining name-mangling techniques that would prevent the name collisions that, otherwise, would inevitable occur with downward HOFs. However, students never really understood these limitations as, from their point of view, there was nothing problematic with HOFs.

In order to improve the learning process, the only option is to use a programming language where HOFs are as natural as normal functions. For this reason, in the last few years we decided to stop teaching using AutoLISP and we now teach using Racket. This move tremendously simplified the explanation of HOFs, as they now can be presented without any caveats and students can use them without any limitations.

CONCLUSIONS

The expressive power of a language measures the breadth of ideas that can be described. Throughout the history of mankind, we have invented numerous ways of increasing the expressive power of our languages. The concept of HOF is one of those inventions that had a transformative effect in our ability to describe nature and, as a result, have been used in many different areas, including genetic programming (Binard 2008), constructive solid geometry (Davy 1995), and shape grammars (Lewis et al. 2004).

In this paper we explained the expressive power of HOFs and the potential of its use in GD. We illustrated that potential by presenting a 3D model of The Market Hall entirely generated by a GD program. This program was developed by one student and it extensively explores the concept of HOF, allowing the simplification of the code and reducing the necessary effort to produce it.

In order to take advantage of HOFs, it is necessary to know the strengths and limitations of programming languages, particularly, regarding the support for HOFs. Otherwise, the effort to overcome the language limitations not only distracts from the main programming goals but might also overcome the benefits of the use of HOFs. The support for HOFs and the increased expressiveness they allow are, thus, strongly dependent on the specific programming language that is being used. In this paper, we considered the most used programming languages in the GD field and we analyzed their support for higher-order programming. Table 1 synthesizes our findings, showing that Phyton, PLASM, and Racket are the languages that better support higher-order programming.

Table 1
Programming language support for higher-order-programming in the GD area. A solid circle means that there is effective support. A half-circle means that there is some support but it might require additional efforts from the programmer. A cross means that there is no support.

	Python	AutoLISP	VBScript	Grasshopper	GDL	MAXScript	Processing	PLaSM	Racket
Paradigm									
Functional	●	●				●		●	●
Imperative	●	●	●		●	●	●		●
Object oriented	●		◐			◐	●		●
Dataflow				●					
Scope									
Lexical	●		●		●	●	●	●	●
Dynamic		●							●
HOFs									
Downward	●	◐	✕	✕	✕	●	✕	●	●
Upward	●	✕	✕	✕	✕	✕	✕	●	●
Using Eval		◐	◐	◐	✕		✕		

In our teaching experience, the use of languages that only provide partial support for HOFs, such as Au-

toLISP, forces the teacher to waste precious time explaining limitations that are difficult to understand and accept by the students. We believe that higher-order programming should be considered a natural extension of programming with functions and, consequently, teaching the definition and use of HOFs should be a natural extension of teaching the definition and use of functions. This is only possible, however, when the programming language being used fully supports HOFs, a consideration that made us move from teaching with AutoLISP to teaching with Racket, allowing us to drastically simplify the explanation of HOFs and saving time for discussing more interesting uses of HOFs.

ACKNOWLEDGEMENTS

We would like to thank the architect Rita Fernandes for sharing with us the GD program for the Market Hall project.

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

REFERENCES

Barendregt, HP 1984, *The Lambda Calculus*, North-Holland, Amsterdam

Binard, F and Felty, A 2008 'Genetic programming with polymorphic types and higher-order functions', *Proceedings of the 10th annual conference on Genetic and evolutionary computation (GECCO '08)*, Atlanta, pp. 1187-1194

Boranyak, S 2010, 'Archetype', *ASCE*, 80(2), pp. 76-79

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

Felleisen, M 1991 'On the expressive power of programming languages', *Selected papers from the symposium on 3rd European symposium on programming (ESOP '90)*, Amsterdam, pp. 35-75

Fry, B and Reas, C 2010, *Getting Started with Processing*, O'Reilly Media

Hughes, J 1989, 'Why functional programming matters', *The computer journal*, 32(2), pp. 98-107

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

Leitão, A, Cabecinhas, F and Martins, S 2010 'Revisiting the Architecture Curriculum: The programming perspective', *Proceedings of 28th eCAADe*, Zurich, pp. 81-88

Lewis, J, Rosenholtz, R, Fong, N and Neumann, U 2004 'VisualIDs: automatic distinctive icons for desktop interfaces', *ACM SIGGRAPH 2004*, New York, pp. 416-423

Lopes, J and Leitão, A 2011 'Portable Generative Design for CAD Applications', *Proceedings of ACADIA 2011*, Alberta, pp. 196-203

McCormack, J, Dorin, A and Innocent, T 2004 'Generative design: a paradigm for design research', *Proceedings of Futureground*, Melbourne

Moses, J 1970 'The function of FUNCTION in LISP or why the FUNARG problem should be called the environment problem', *ACM Sigsam Bulletin*, pp. 13-27

Paoluzzi, A and Sansoni, C 1992, 'Programming language for solid variational geometry', *Computer-AidedDesign*, 24(7), pp. 349-366

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

Tobin-Hochstadt, S 2011, 'Languages as libraries', *ACM SIGPLAN Notices*, 46(6), pp. 132-141

Watson, A (eds) 2009, *GDL handbook: A comprehensive guide to creating powerful ArchiCAD objects*, Cadimage Solutions, NewZealand