Title
IMPERATIVE/FUNCTIONAL/OBJECT-ORIENTED

Authors
Steinfeld, Kyle
Olascoaga, Carlos Emilio Sandoval

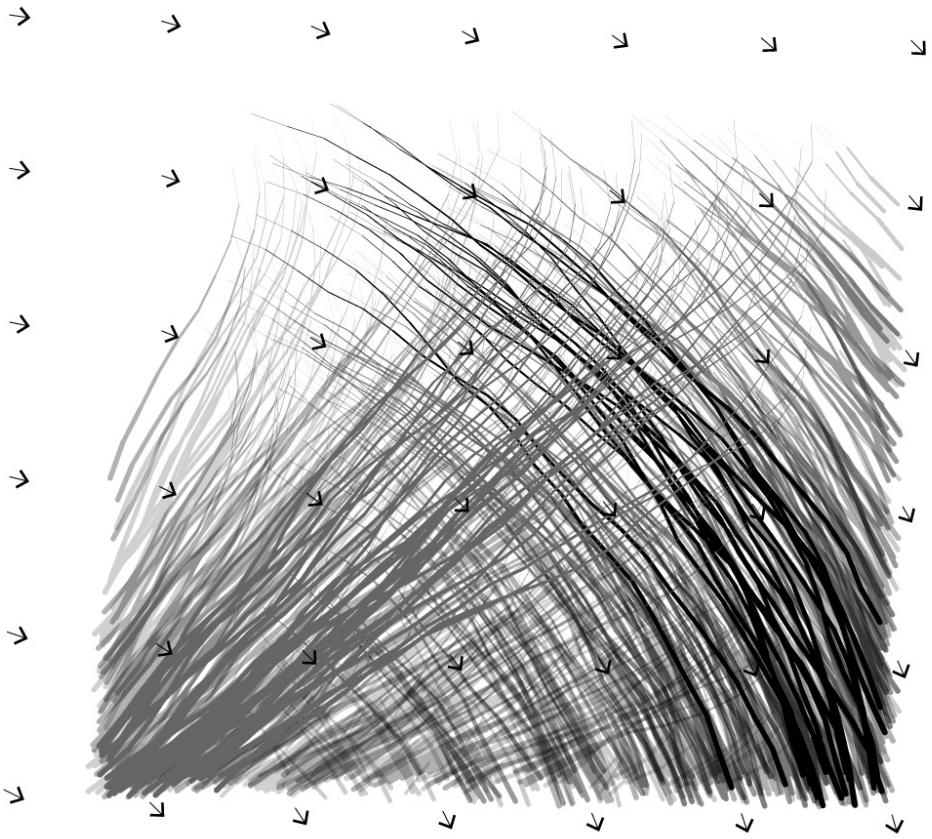
Publication Date
2014

Peer reviewed

IMPERATIVE/ FUNCTIONAL/OBJECT-ORIENTED

AN ALTERNATIVE ONTOLOGY OF PROGRAMMATIC PARADIGMS FOR DESIGN

Kyle Steinfeld
UC Berkeley
Carlos Emilio Sandoval Olascoaga
Massachusetts Institute of Technology



ABSTRACT

Distinctions between approaches to programming for design applications are marked by the split between Visual Programming Languages (VPLs) and Textual Programming Languages (TPLs).[1] While this distinction has proven useful in characterizing the applicability of programming languages to design applications, it struggles to address languages that hybridize visual and textual modes, and cannot account for other structural features beyond user interface. An alternative ontology, differentiated by programmatic paradigm[2] suggests an improved method of assessment. This study applies a programmatic paradigm taxonomy to two programming environments: Decodes and DesignScript. The former is a domain-specific TPL that exhibits qualities of an Imperative Programing Language (IPL) and an Object-Oriented Programming Language (OOPL). The latter, a VPL-TPL hybrid allows users to move between IPL, OOPL, and Functional Programming Language (FPL) modes. Proceeding through the analysis of case studies, this study yields a set of guidelines for the application of each of these paradigms.

INTRODUCTION
PROGRAMMATIC PARADIGMS

Programming paradigms are defined by the methods and structures used to develop a program. While different programming paradigms have emerged through time, two distinct paradigms, Declarative and Imperative, stand in stark contrast. While an imperative paradigm structures a program as a set of instructions to solve a problem in different states, a declarative paradigm solves a problem based on its description, with language-specific implementations. A number of further programming paradigms exist within and outside the range of declarative and imperative paradigms, most of them exhibiting characteristics and features of multiple programming paradigms. Modern programming languages generally rely on a combination of four paradigms: functional, imperative, object-oriented, and logical.[3] Following a rapid expansion of the quantity and variety of programming languages in the 1990s, the utility of these divisions has been called into question. Modern programming languages structures' are rarely based on a single paradigm, and instead allow programmers to select the most appropriate method for a given problem. While some of these paradigms combine features of both Declarative and Imperative modes, for example, each proceeds through a unique set of methods, structures and terms. The distinctions offered by the archetypal paradigm described below offer a potentially useful roadmap to understanding the use of computer programming in design. Of the general programming paradigms, only three are regularly employed in a design context:

Imperative Programming Languages (IPL) are the most common programming paradigm used in mainstream languages. An *IPL* workflow can be understood as a series of steps executed in a defined order, often described as an algorithm. "Flow Control" is a fundamental characteristic of an *IPL*, and is provided by loops, "if/else" statements and other similar structures. In this way, the basic operations in an imperative language are to perform, replicate tasks in sequence, and track state changes of the program as this process unfolds.

Functional Programming Languages (FPL), rather than defining a series of steps to be performed and state changes, define a set of interrelated operations. While the logic of a program is defined through flow control in *IPL*, in *FPL* the logic of a program is solely described through the algorithm. In other words, *FPL* describes what the program should do, while an *IPL* describes how the program should do it. Rather than basing a computation on the state of a program, functional programming focuses on the evaluation of mathematical expressions, where the results are dependent on the function's inputs. In functional programming, execution order is a by-product of a set of topological relationships defined between functions. Most *Visual Programming Languages*, including the popular *Grasshopper* programming environment, exhibit characteristics of an *FPL*.

Object Oriented Programming Languages (OOPL) focus on the structure of data rather than transformations of that data, and are most often employed to encapsulate and modularize systems. An *OOPL* allows the integration of code and data into a single object: an abstract data type with fields or properties describing the object, and methods or operations for the objects. Object orientation facilitates modularization and extensibility, encouraging end-users to customize and extend more primitive languages.

PROGRAMMING LANGUAGES ADDRESSED IN THIS STUDY

Python, created in the late 1980s by Guido van Rossum, is a high-level interpreted, non-compiled, multi-paradigm language that combines elements of an *IPL* and *OOPL*. Python is intended to be a minimalist and highly readable language, favoring style over complexity. Designed with a basic core library and a simplified method for defining modules, it is easily extensible by end-users through dynamic typing.

Decodes, created by Kyle Steinfeld in 2013 [4], is a platform-independent computational geometry library written for Python that includes a range of features specific to the domain of architectural design. This language forms the basis of all of the examples explored in the case study section of this paper.

An *Associative Programming Language (APL*), as defined by the creators of *DesignScript*, is a multi-paradigm programming language, combining Imperative, Object Oriented, and Functional paradigms in a specific way, with additional functionality provided by actions termed 'replication' and 'modifiers'. As implemented within *DesignScript*, code blocks are allowed to be defined as *functional* or *imperative* modes, enabling to switch between paradigms. Functional blocks are the default mode and are termed as "Associative" blocks within the software. For simplicity, we will hereupon refer to these two types as "associative" vs "imperative" blocks, following the convention found in previous writing.[5] The creators of *DesignScript* argue that the combination of topological transformations with replication mechanisms and modifiers constitute a more appropriate generative tool for the design domain.

LITERATURE REVIEW

Many recent studies of the application of programming to design have focused upon distinctions of user interface, in particular the relative merits of visual programming and textual programing in a design context. In this section, we summarize the prevailing comparisons of visual and textual programming in a design context, and discuss how *DesignScript* disrupts this landscape, presenting itself as a hybrid language.

VISUAL VS TEXTUAL

The introduction of *VPLs* and *TPLs* in design software has provided designers the capacity of understanding, entering, and modifying the inner workings of design software through programming, allowing designers to customize software and provide flexibility. The intuitive nature and direct feedback allowed by *VPLs* make them easy to learn tools; a property that has helped to make *VPLs* (most notably, McNeel's *Grasshopper*) among the most popular tools for generative design. Additionally, *VPLs* allow for an abstraction and diagrammatic representation of the design process, making them powerful tools in design pedagogy. While *VPLs* are good entry tools for designers, they pose scalability and complexity limitations.[6] The structure and characteristics of *TPLs*, in contrast, make them well suited for more complex operations, allowing modularization, control mechanisms, and a multi-paradigm approach to programming. Additionally, *TPLs* present more possibilities for the development and implementation of generative systems, going beyond the linear parametric variation[7] supported by *VPLs*.

DESIGNSCRIPT–A NEW PARADIGM?

As an exploratory, domain-specific design tool, *DesignScript*, combines conventional modeling, associative modeling, and programming, situating it at the midpoint of *VPLs* and *TPLs*.[8] *DesignScript* provides an associative language, which represents the flow of data in a human readable text notation,[9] resembling the graph dependencies of *VPLs*, and combines it with conventional programming capabilities. By combining both approaches to computational design, it is accessible to both novice and expert programmers. *DesignScript* is defined as a domain-specific language, providing methods to support geometric manipulation and representation appropriate to an architectural context, and supports simplified syntax rules, reducing restrictions associated with general programming languages. Its limited host-independency (it is able to communicate with a range of Autodesk software) allows a multi-disciplinary approach and enables different stages of design process–from design exploration to simulation and analytics.

Seen through the lens of programmatic paradigm, *DesignScript* is a hybrid programming language, combining object-oriented, imperative, and functional languages with domain-specific concepts. By combining the different programming paradigms, it supports replication operations common in Associative paradigms, and by allowing the user to switch into an Imperative paradigm at the same time enables specific iterative differentiation common in *IPL*. While flow control in imperative programming is provided by mechanisms such as loops and if-else statements, associative programming uses a concept of associative dependencies inherent within each statement to create a topological ordering of statements. It is asserted that this specific combination of features selected from *FPL* and *OOPL*, and the possibility to intermix an *IPL* within the code represents a new programmatic paradigm that the authors of *DesignScript* term Associative Programming, a paradigm that better reflects the typical modeling operations found in architectural design.

METHODS

Discussed here are the structural and syntactical differences between *IPL*s and *APL*s most impactful in generative design, the rationale and practicalities of implementing design domain tasks in both paradigms, and the relative merits of each language in a selection of specific situations.

The study is presented in the context of a residency in the Autodesk IDEA Studio.[10] Supported by this residency program, the authors re-implemented in *DesignScript* over fifty computational geometry lessons and code samples that had been previously written for Decodes. The specific case studies presented here are selected from lessons and examples developed for a forthcoming book,[11] in contract to be published in 2015.

NOTABLE FEATURES OF SELECTED PROGRAMMING LANGUAGES

The contradictory features of *IPL*s and *APL*s facilitate certain ways of working in a design context, and discourage others. At times the contrast is acute, wherein one paradigm directly facilitates functionality where another forbids it (see object mutability, below). Other times the contrast is more subtle, wherein both paradigms allow for similar approaches to be taken, with one strongly supporting it while the other merely allows it, sometimes encumbering the user with complicated syntax (see list comprehension vs replication, below)

The following account of contrasting features and approaches to similar programming tasks is derived both in anticipation of and reflection upon domain-specific application, as discussed in the

case-studies section, below. It was derived both as a report of opportunities seen and challenges faced during training on the *APL*, and upon reflection after re-implementing examples previously developed using the *IPL*.

OBJECT MUTABILITY (*IPL*) VS GRAPH MAPPING (*APL*)

When extending the *IPL* by creating new objects, it is possible to continuously add new properties and modify the object after it is constructed, a condition known as mutability of a class.[12] Mutability of classes becomes useful when certain properties are not required for constructing an instance, but would be useful when later added by private methods. A common use-case of this feature may be found in class inheritance. In an *IPL*, a class can be constructed as a subclass of another class or classes, inheriting their properties and methods. Following the mutability principle, a subclass can also modify the parent class if needed. In a design domain context, object mutability simplifies workflow by allowing an exploratory approach to object-oriented programming (not adding a property until it is needed), by encouraging a more relaxed syntax, and by allowing the user extension of a given type.

Differently, objects are immutable in an *APL* due to the graph structure of the language. Once an object is created and mapped to the graph, it is impossible to transform the object's properties. While object immutability can provide security by avoiding the modification of an object by external agents, it can stymie the exploratory nature of the design process. In such a case, all the properties would have to be defined a-priori via the class constructor. While the graph's topological order and relationships would be constantly updating each variable, unless a property is initially defined within the constructor, the object won't change.

FUNCTIONS (IPL) VS MODIFIERS (APL)

As implemented in *DesignScript*, an *APL* incorporates the concept of modifiers, which allow for the creation of transformation blocks modifying an object, avoiding the creation of a new variable for every transformation operation. Modifiers facilitate common modeling operations of repetition and variation of elements throughout a field.

In contrast, geometric transformations in an *IPL* often rely on static functions that return a new (modified) instance of an object, instead of modifying the original instance in-place. In order to avoid redundancy in naming operations and variables, it is possible to chain expressions together in an *IPL*, performing the set of transformation operations in a single expression. The original instance of the object can thusly be rewritten with a new transformed instance, as seen in (Figure 3).

```
class AmmannA3Tile(object):
 def __init__(self,xf=Xform(), lineage="RT",scale=-
None):
 self.lineage = lineage
 self.xf = xf
 self._xf_scale = Xform.scale(1/TAU) #0.618033
 self.scale = scale

 def _cs_from_base_pts(self,pt_o=0,pt_x=1,pt_y=2):
 pt_0 = self._base_pts[pt_o]
 pt_x = self._base_pts[pt_x]-self._base_pts[pt_o]
 pt_y = self._base_pts[pt_y]-self._base_pts[pt_o]
 return CS(pt_0, pt_x, pt_y)

 # world base Points for this tile
 @property
 def base_pts(self):
 return [p*self.xf for p in self._base_pts]

 # draw a PGon from the Points
 def to_pgon(self):
 pg = PGon(self.base_pts[:self.boundary_base_pt_
cnt])
 pg.name = self.lineage
 return pg

class AmmannA3TileA(AmmannA3Tile):
 # the idealized base Points for all Tiles of type A
 @property
 def _base_pts(self):
 return [
 Point(0.0, 0.0),
 Point(TAU_3, 0.0),
 Point(TAU_3, TAU_2),
 Point(TAU_3-TAU_2, TAU_2),
 Point(TAU_3-TAU_2, TAU),
 Point(0, TAU)
 ]

 def inflate(self):
 xf_pos = self._cs_from_base_pts(0,1,5).xform
 b0 = AmmannA3TileB(self.xf * xf_pos * self._xf_
scale,
 self.lineage+",b0")

 xf_pos = self._cs_from_base_pts(1,2,0).xform
 a0 = AmmannA3TileA(self.xf * xf_pos * self._xf_
scale,
 self.lineage+",a0")
 return [b0,a0]

 @property
 def boundary_base_pt_cnt(self): return 6
```

1 Object mutation in IPL (Sandoval, Steinfeld, 2014)

```
p = {
 Circle.ByCenterPointRadius(Point.ByCoordi-
nates(0,0,0), 5.0);
 Translate(40, 0, 0);
 Transform(CoordinateSystem.Identity(),
 CoordinateSystem.BySphericalCoordinates(
 CoordinateSystem.Identity(), 10, 15, 25));
};
```

2 An object modifier in DesignScript (Sandoval, Steinfeld, 2014)

```
def modifier(geometry):
 geometry *= Xform.translation(
 Vec(40)*Xform.change_basis(CS(),-
CylCS(Point(10,1,5))))
 return geometry
a = modifier(Circle(Plane(), 5.0))
```

3 A function in Decodes / Python (Sandoval, Steinfeld, 2014)

```
# Imperative
array1 = [0,1,2]
list = []
for i in array1:
 list.append(i * 2)

print list
>> [0,2,4]


# Imperative (list comprehension)
array1 = [0,1,2]

print [i * 2 for i in array1]
>> [0,2,4]
```

4 Loops and List Comprehension in Decodes / Python (Sandoval, Steinfeld, 2014)

```
//Associative
array1 = {0,1,2};

Print(array1 * 2);
>> {0,2,4}
```

5 Replication in DesignScript (Sandoval, Steinfeld, 2014)

LOOPS AND LIST COMPREHENSIONS (*IPL*) VS REPLICATION (*APL*)

Associative programming introduces the concept of "replication": the ability to interchangeably use collections and single values, a feature that provides significant flexibility to the designer. When a collection is passed to a functional component, the function is executed once for each given element in the collection, simplifying the propagation of common design operations. The topological relationships characteristic of associative programming allow for transformations and changes between different data types, from single elements to collections, when a variable is modified.

"List Comprehensions", a functional programming feature in Python, provides a similar abstraction mechanism, enabling the simplification of a script. However, as the syntax involved can be cumbersome, they are limited to low-dimensional data structures, and are rarely employed on more than two-dimensional arrays of objects.

Expanding upon this distinction, associative programming simplifies the task of weaving numeric type lists together using "Zipped Replication". Building upon a series of replication operations, when a number of collections are matched the corresponding elements are evaluated or "woven" together using the syntax seen below.

Nested 'for' loops are similarly simplified and structured with "Cartesian Replication": when two or more collections are matched together, all members of the collections are evaluated by cross-referencing them with every member of the collections in "N" dimensions.

In this way, it is possible to define the sequence of the replications occurring with the 'replication guides', making it possible to easily modify the structure of the data.

CASE STUDIES

In this section, we present an analysis of a selection of case studies, chosen from a body of examples both pedagogical and derived from practice. These case studies are taken from a larger collection of code samples fully detailed in a forthcoming book, The Architect's Field Guide to Computation (working title)[13] in contract to be published by Routledge in 2015. See www.decod.es for list of all case study examples translated, including notes on the relevant *APL*/*IPL* comparisons drawn from each.

CASE STUDY 1: A DESIGN SPACE OF COLORS

A simple color space exploration demonstration, this example allowed for familiarization with a particular feature of the *APL*: replication. A three-dimensional design-space representation of individual colors is constructed based on the combination of lower-dimensional collections. In the context of an *IPL*, the exploration relies on nested loops. Differently, an *APL* approach takes advantage of "Cartesian Replication"; it first replicates a given operation a number of times by passing a collection, and then interrelates multiple collections with each other based on "Cartesian Guides". This approach enables the intuitive management and development of simple data structures, allowing different combinations between collections based on the modification of the "Cartesian Guides".

```
#Imperative
array1 = [0,1,2]
array2 = [3,4,5]

list = []
for i, element in enumerate(array1):
 list.append(element + array2[i])

print list
>> [3,5,7]

#Imperative (list comprehension)
array1 = [0,1,2]
array2 = [3,4,5]

print [element + array2[i] for i, element in enu-
merate(array1)]
>> [3,5,7]
```

6 Imperative list combination (Sandoval, Steinfeld, 2014)

CASE STUDY 2: A DESIGN SPACE OF CURVES

Curves can be parameterized and geometrically expressed via mult*IPL*e mathematical forms. In the context of the *IPL*, curve objects were developed using parametric equations: they are the result of vector / point functions replicated within interval of values. Essentially, the curve function is replicated by looping throughout the interval of values creating an array of points.

The example develops two different mathematical curve functions, and interpolates values between them, stacking them vertically to resemble tower-plate configurations. In the context of the *IPL* a nested loop is used to generate the curves and define their vertical location. While in the context of an *APL*, Cartesian Replication is used to perform 3D combinations between collections, defining the curves and their vertical position with a single operation. Further exploration of the *APL* approach demonstrated advantages in the easy transformation of the resulting collections' data structure, thus allowing different curve configurations that vary between vertical, radial, and diagonal.

```
//Associative
array1 = {0,1,2};
array2 = {3,4,5};

Print(array1 + array2);
>> {3,5,7}
```

7 Associative zipped replication (Sandoval, Steinfeld, 2014)

```
#Imperative
array1 = [0,1,2]
array2 = [0,1,2]

list = []
for i in array1:
 for n in array2:
 list.append(Point(i,n))

print list
>> [Point(X=0, Y=0, Z=0), Point(X=0, Y=1,
Z=0),Point(X=0, Y=2, Z=0)
,{Point(X=1, Y=0, Z=0), Point(X=1, Y=1, Z=0),
Point(X=1, Y=2, Z=0)
,{Point(X=2, Y=0, Z=0), Point(X=2, Y=1, Z=0),
Point(X=2, Y=2, Z=0)]
```

8 Imperative nested list combination (Sandoval, Steinfeld, 2014)

CASE STUDY 3: FLOCKING

As a simple flocking code, this example allowed the exploration of the graph structure features of the *APL*. Traditional flocking examples are based on the capacity of individual agents to be aware of other agents. In the context of an *IPL*, flow control through conditional statements is employed for context awareness. Through the associative relationships of variables in the *APL*, it is possible to define graph-like relationships between the agents, thereby eliminating the need for flow control structures. While it is possible to extend the *IPL* and the *APL*, the *APL*'s graph structure make objects immutable; an immutable object forces the designer

```
//Associative
array1 = {0,1,2};
array2 = {0,1,2};

Print(Point.ByCoordinates(array1<1>,array2<2>,0));
>> {{Point(X=0, Y=0, Z=0), Point(X=0, Y=1, Z=0),
Point(X=0, Y=2, Z=0)}
,{Point(X=1, Y=0, Z=0), Point(X=1, Y=1, Z=0),
Point(X=1, Y=2, Z=0)}
,{Point(X=2, Y=0, Z=0), Point(X=2, Y=1,
Z=0),Point(X=2, Y=2, Z=0)}}
```

9 Associative Cartesian Replication (Sandoval, Steinfeld, 2014).

```
#Imperative
array1 = [0,1,2]
array2 = [0,1,2]

list = []
for i in array1:
 sublist = []:
 for n in array2:
 sublist.append(Point(i,n))
 list.append(sublist)

print list
>> [[Point(X=0, Y=0, Z=0), Point(X=0, Y=1, Z=0),
Point(X=0, Y=2, Z=0)]
,[Point(X=1, Y=0, Z=0), Point(X=1, Y=1, Z=0),
Point(X=1, Y=2, Z=0)]
,[Point(X=2, Y=0, Z=0), Point(X=2, Y=1, Z=0),
Point(X=2, Y=2, Z=0)]]
```

10 Imperative nested structured list combination (Sandoval, Steinfeld, 2014)

to define all object properties, a disadvantage for the exploratory nature of generative design in the domain. Through the development of the example, the object had to be rewritten constantly to accommodate object immutability.

CASE STUDY 4: REACTIVE SPACING OF TOOLPATHS

Similarly to curves, surfaces can be geometrically expressed using parametric equations. In the context of the *IPL*, the construction of a surface requires only a parametric function, but its representation (surrogate) is constructed with 3D nested loops. In the context of the *APL*, Cartesian Replication is used to combine collections in 3D, constructing surfaces without the need of nested loops. At the same time, surface directionality is easily modified with Cartesian Guides.

In an *IPL* a nested loop is used to combine collections and create UV values to evaluate the surface at given parameters. Through the loops, points are structured in a nested 2D collection to create polylines between them. In an *APL* approach, Cartesian Replication evaluates a surface at the given parameters and structures them accordingly with Cartesian Guides.

CASE STUDY 5: PARAMETRIC MODEL OF ICD / ITKE RESEARCH PAVILION

In the context of the *APL*, the code allowed a scalable design exploration, starting with the design of a single element. Once a single element was properly defined, the replication functionality allowed increasing the scale and number of elements computed without rewriting the operations, by passing a collection, instead of a single element to the initial code.

In the context of the *IPL*, the initial code and explorations developed for a single element had to be rewritten, packaged and modularized when scaling up to multiple elements. The example showed the limitations of a purely functional approach within the *APL*. While it is possible to easily replicate a function that requires a single variable as an input, it is not possible to weave multiple lists together into a new multi-dimensional collection in order to replicate functions that require multiple parameters as inputs. Within the *APL*, it is possible to use *IPL* code blocks to complement the functional approach.

DISCUSSION

Upon reflection, the case study described above yields a set of suggested guidelines for the context-appropriate application of the *IPL* and *APL* programming paradigms. These guidelines are impactful to

design pedagogy and practice, both as a way of choosing appropriate programming languages for a given task, and as a use guide for programming environments (such as *DesignScript*) that hybridize the two approaches. In such a hybrid environment, one may selectively apply imperative and associative models. The guidelines found below may assist the user of such hybrid software in the selection of *IPL* and *APL* features of the language.

GUIDELINE ONE–APPROPRIATE USES OF REPLICATION

An *APL* largely simplifies replicating operations throughout collections of objects. Replication mechanisms allow an easy propagation and interrelation of single or multiple collections, providing the designer more control over shaping the resulting data structures. Anytime a designer is exploring different collection data structures by modifying their relationships, an *APL* will enable a clearer syntax and code in comparison to an *IPL*'s "nested loops" strategy. Replication operations in nested collections that are common to the architectural design domain can be easily implemented with an *APL* (up to three-dimensions). Similarly, the relationships between the resulting data structures can be controlled through "Cartesian Guides", enabling an exploratory coding characteristic of initial design phases.

This technique, however, has its limits: while with an *APL* it is possible to explore a one-dimensional collection of generic objects through replication, producing nested combinations between collections in this way is limited to collections of numeric types. In this case, an *IPL* would be preferred, and is able to operate on any object type. Similarly, an *IPL* allows "weaving" or combining mult*I-PL*e collections into a new collection.

GUIDELINE TWO–CONDITIONALS

While the *APL* allows some degree of flow control with the implementation of "in-line conditionals", equivalent to an "if/else" statement, more intricate flow control is more cumbersome. At the same time, *APL*'s "in-line conditionals" are limited to a single statement and cannot be combined with additional conditional statements. Whenever flow control through conditional statements requires more than a binary evaluation, or a conditional statement modifies multiple variables or further conditional statements, an *IPL* should be used.

GUIDELINE THREE–DESIGN EXPLORATION THROUGH TYPE DEFINITION

While both *IPL* and *APL* languages allow user extension through dynamic typing, the *APL*'s graph structure does not allow object

```
//Associative
array1 = {0,1,2};
array2 = {0,1,2};

Print(Point.ByCoordinates(array1<2>,array2<1>,0));
>> {{Point(X=0, Y=0, Z=0), Point(X=1, Y=0, Z=0),
Point(X=2, Y=0, Z=0)}
,{Point(X=0, Y=1, Z=0), Point(X=1, Y=1, Z=0),
Point(X=2, Y=1, Z=0)}
,{Point(X=0, Y=2, Z=0), Point(X=1, Y=2, Z=0),
Point(X=2, Y=2, Z=0)}}
```

11 Associative Cartesian Replication (Sandoval, Steinfeld, 2014).

mutability. When exploring a design through the definition of new object types, specific properties often need to be added dynamically after the object has been constructed. In this use case, an *IPL* should be used. This includes extending an object through a subclass; with an *APL*, the subclass will only share the same properties of the parent class.

GUIDELINE FOUR – RECURSIVE STRUCTURES

Recursive solutions, where the result of an operation depends on previous instances of the operation, generally depend on defining loops. An *APL* relying only on the replication of function calling increases the challenge and complexity of recursive operations. Furthermore, nested recursive operations cannot be implemented by the *APL*. For ease of implementation, designs that rely heavily on recursive operations, such as fractals, should be performed with an *IPL*.

IMPACT AND FUTURE WORK

The comparison between the implementation in an *APL* and an *IPL* of the selected case studies has yielded a number of suggested guidelines for an application of each paradigm within the domain of study. Expanding this selection of case studies will allow the exploration and comparison between both paradigms; the graph structure characteristic of the *APL* has a particular potential for future study. While most of the selected case studies employ the graph structure to propagate transformations and operations across the nodes of the graph, a large potential lies in defining graph relationships between the objects replacing *IPL*'s flow control. Similarly, future case study implementations hold the potential to determine the application of the *APL*'s graph structure for analysis and optimization operations commonly encountered in the domain.

Features of both *TPLs* facilitate certain ways of working within a paradigm and sometimes forbid a particular functionality. As the *APL* investigated here, *DesignScript* is quite new, new applications are likely to be discovered as the language continues to develop. In particular, the ability to combine collections of any type would allow the incorporation of the clarity and syntax simplicity characteristic of the *APL* into the design process and present further incentive to this framework over an *IPL*. In the context of the residency in the Autodesk IDEA Studio, the case studies were implemented the *APL*'s textual IDE. Current development work focuses on merging the *APL* with the VPL "Dynamo". Once completed, the hybrid *VPL-TPL* will extend the abstraction mechanisms of the *APL*, unveiling further domain specific implementation guidelines.

ACKNOWLEDGMENTS

This work was supported as part of the Autodesk IDEA Studio Residency Program in San Francisco, CA.

NOTES

1. António Leitão, Luís Santos, and José Lopes, "Programming Languages For Generative Design: A Comparative Study", International Journal of Architectural Computing 10, no. 1 (March 2012): 139–62, doi:10.1260/1478-0771.10.1.139.

2. Peter Van Roy and Seif Haridi, "Concepts, Techniques, and Models of Computer Programming", MIT Press (2004).

3. Shriram Krishnamurthi, "Teaching Programming Languages in a Post-Linnaean Age", SIGPLAN Not. 43, no. 11 (November 2008): 81–83, doi:10.1145/1480828.1480846.

4. Kyle Steinfeld, "Decodes: A Platform-Independent Computational Geometry Environment", in Open Systems: Proceedings of the 18th International Conference on Computer-Aided Architectural Design Research in Asia (2013), ed. R Stouffs, P Janssen, and B Tuncer (presented at the CAADRIA 2013, Singapore, 2013), 499–508.

5. Robert Aish, "DesignScript: Origins, Explanation, Illustration," in Proceedings of the Design Modeling Symposium Berlin 2011, vol. 3 (presented at the Computational Design Modeling - Design Modeling Symposium Berlin 2011, Berlin: Springer Berlin Heidelberg, 2011), 1–8, doi:10.1007/978-3-642-23435-4_1.

6. Celani and Vaz, "CAD Scripting And Visual Programming Languages For Implementing Computational Design."

7. Robert Woodbury, "Elements of Parametric Design" (Routledge, 2010).

8. Aish, "DesignScript: Origins, Explanation, Illustration."

9. Robert Aish, "DesignScript: A Learning Enviroment for Design Computation", in Proceedings of the Design Modeling Symposium Berlin 2013 (presented at the Computational Design Modeling - Design Modeling Symposium Berlin 2013, Berlin: Springer Berlin Heidelberg, 2013).

10. "Autodesk IDEA Studio Residency Program," accessed April 20, 2014, http://www.autodesk.com/gallery/idea-studio.

11. Kyle Steinfeld and Joy Ko, "The Architect's Field Guide to Computation (Working Title)" (Routledge, 2015).

12. Brian Goetz et al., Java Concurrency in Practice, 1 edition (Upper Saddle River, NJ: Addison-Wesley Professional, 2006).

13. Kyle Steinfeld and Joy Ko, "The Architect's Field Guide to Computation (Working Title)".

REFERENCES

António Leitão, Luís Santos, and José Lopes. 2012. "Programming Languages For Generative Design: A Comparative Study", International Journal of Architectural Computing 10, no. 1: 139–62, doi:10.1260/1478-0771.10.1.139.

Aish, Robert. 2011. "DesignScript: Origins, Explanation, Illustration," in Proceedings of the Design Modeling Symposium Berlin 2011, vol. 3 (presented at the Computational Design Modeling - Design Modeling

Symposium Berlin 2011, Berlin: Springer Berlin Heidelberg, 2011), 1–8, doi:10.1007/978-3-642-23435-4_1.

Aish, Robert. 2013. "DesignScript: A Learning Enviroment for Design Computation", in Proceedings of the Design Modeling Symposium Berlin 2013 (presented at the Computational Design Modeling - Design Modeling Symposium Berlin 2013, Berlin: Springer Berlin Heidelberg.

Brian Goetz et al. 2006. Java Concurrency in Practice, 1 edition, Upper Saddle River, NJ: Addison-Wesley Professional.

Gabriela Celani and Carlos Eduardo Verzola Vaz. 2012. "CAD Scripting And Visual Programming Languages For Implementing Computational Design", International Journal of Architectural Computing 10, no. 1: 121–38, doi:10.1260/1478-0771.10.1.121.

Krishnamurthi, Shriram. 2008. "Teaching Programming Languages in a Post-Linnaean Age", SIGPLAN Not. 43, no. 11: 81–83, doi:10.1145/1480828.1480846.

Peter Van Roy and Seif Haridi. 2014. "Concepts, Techniques, and Models of Computer Programming", MIT Press.

Steinfeld, Kyle. 2013. "Decodes: A Platform-Independent Computational Geometry Environment", in Open Systems: Proceedings of the 18th International Conference on Computer-Aided Architectural Design Research in Asia (2013), ed. R Stouffs, P Janssen, and B Tuncer (presented at the CAADRIA 2013, Singapore, 2013), 499–508.

Steinfeld, Kyle, and Ko, Joy. 2015. "The Architect's Field Guide to Computation (Working Title)", Routledge.

Woodbury, Robert. 2010. "Elements of Parametric Design", Routledge.

IMAGE CREDITS

Figure 1. Sandoval, Carlos and Steinfeld, Kyle (2014) Object mutation in IPL

Figure 2. Sandoval, Carlos and Steinfeld, Kyle (2014) An object modifier in DesignScript

Figure 3. Sandoval, Carlos and Steinfeld, Kyle (2014) A function in Decodes / Python

Figure 4. Sandoval, Carlos and Steinfeld, Kyle (2014) Loops and List Comprehension in Decodes / Python

Figure 5. Sandoval, Carlos and Steinfeld, Kyle (2014) Replication in DesignScript

Figure 6. Sandoval, Carlos and Steinfeld, Kyle (2014) Imperative list combination

Figure 7. Sandoval, Carlos and Steinfeld, Kyle (2014) Associative zipped replication

Figure 8. Sandoval, Carlos and Steinfeld, Kyle (2014) Imperative nested list combination

Figure 9. Sandoval, Carlos and Steinfeld, Kyle (2014) Associative Cartesian Replication

Figure 10. Sandoval, Carlos and Steinfeld, Kyle (2014) Imperative nested structured list combination

Figure 11. Sandoval, Carlos and Steinfeld, Kyle (2014) Associative Cartesian Replication

Figure 12. Sandoval, Carlos and Steinfeld, Kyle (2014) Imperative code for building curves

Figure 13. Sandoval, Carlos and Steinfeld, Kyle (2014) Similarly, curves can be built using parametric equations in an APL, using the replication mechanism

KYLE STEINFELD Assistant Professor specializing in digital design technologies in the Department of Architecture at UC Berkeley, is the author of the "Architect's Field Guide to Computation", in contract with Routledge to be published in 2015, and the creator of Decod.es, a platform-agnostic geometry library, and a collaborative community that promotes computational literacy in architectural design. He teaches undergraduate and graduate design studios, core courses in architectural representation, and advanced seminars in digital modeling and visualization. Professionally, he has worked with and consulted for a number of design firms, including Skidmore Owings and Merrill, Acconci Studio, Kohn Petersen Fox Associates, Howler/Yoon, Diller Scofidio Renfro, and TEN Arquitectos. His research interests include collaborative design technology platforms, design computation pedagogy, and bioclimatic design visualization. He holds a Masters of Architecture from MIT and a Bachelor's Degree in Design from the University of Florida.

CARLOS SANDOVAL OLASCOAGA is an architect that specializes in computational design. Before joining the Computation Group at MIT, he was a researcher at UC Berkeley, and a consultant at the Data Lab at UC Berkeley, investigating the convergence of design, data science, and social sciences. In the past four years, Carlos has been a lecturer at UC Berkeley and a Visiting Professor at UNAM, and has taught computational design seminars and workshops in the United States, Italy, China and Mexico. His work has been supported by numerous fellowships, more recently by the IDEA Studio at Autodesk, the National Council of Science and Technology, and the Jumex Foundation for Contemporary Arts. He is interested in urban data visualization, computation in design cognition, and technological platforms for design collaboration. Carlos holds a Master of Architecture from UC Berkeley, and a Bachelor of Architecture with Distinction from UNAM in Mexico.