Higher-Order Functions in Aesthetic EC Encodings

James McDermott, Jonathan Byrne, John Mark Swafford, Michael O'Neill and Anthony Brabazon

***Abstract*— The use of higher-order functions, as a method of abstraction and re-use in EC encodings, has been the subject of relatively little research. In this paper we introduce and give motivation for the ideas of higher-order functions, and describe their general advantages in EC encodings. We implement grammars using higher-order ideas for two problem domains, music and 3D architectural design, and use these grammars in the grammatical evolution paradigm. We demonstrate four advantages of higher-order functions (patterning of phenotypes, non-entropic mutations, compression of genotypes, and natural expression of artistic knowledge) which lead to beneficial results on our problems.**

"Writing about music is like dancing about architecture" – various artists.[1]

I. Introduction

In a language with *first-class* functions, a function can be assigned as the value of a variable and passed as an argument to another function (known as *higher-order*). Whether in evolutionary automatic programming or in programming "by hand", this facility is a powerful method of abstraction and generalisation, and allows fine-grained re-use: patterns of re-use which can not be captured by standard automatically-defined functions (ADFs), for example, can be captured using higher-order ADFs (HO-ADFs) [1].

The idea has been used rarely in evolutionary computation (EC), with the exception of seminal work by Yu [2] which demonstrated its advantages on standard non-interactive benchmarks. We now extend the idea to a new representation, grammatical evolution (GE). We also focus on its advantages with emphasis on aesthetic problem domains, discussing **four main advantages**, as described next.

Using higher-order functions can produce more general and generalisable solutions, and by increasing opportunities for re-use, it can **(1) make optimal solutions shorter**, which tends to make them easier to find [2]. It can "enrich" the search space, that is increase the proportion of useful programs. This can help to alleviate a significant problem in interactive EC, the *fitness evaluation bottleneck* [3], [4]. By making more opportunities for re-use, HO-ADFs can lead to phenotypes which are **(2) more highly patterned**, and patterned in different ways, than those characteristic of typical methods of re-use. Such phenotypes are particularly characteristic of aesthetic domains such as music and architectural design. Using HO-ADFs in these two application areas allows us to avoid the common interactive EC (IEC) problem of having to evaluate many random-seeming individuals in early generations. Higher-order representations tend to allow **(3) non-entropic mutations**: a mutation causes multiple consistent changes to the phenotype, rather than a single change which damages the its balance or symmetry. They also present an opportunity for creative workers to **(4) express their artistic knowledge in a natural way**. Many musicians and designers think naturally in terms of patterns, themes with variations, and re-use. Higher-order functions offer one natural and flexible way to express these ideas through the grammar. By contrast, L-systems, cellular automata, and similar biologically-inspired representations tend to produce good *emergent* results that "surprise" the user. This can be an advantage in many situations, but means it is more difficult for the user to introduce a bias to the representation to encourage desired properties in the output.

Our goal in this paper is to demonstrate these properties of higher-order programming and show that they have a beneficial effect not in a particular domain, but in aesthetic domains in general. With this aim of generality, we discuss two very different aesthetic domains, architectural design and music. Both domains are largely aesthetic and subjective, and our claims will be backed by aesthetic examples and arguments, but not by experimental evidence. Future work and papers dedicated to the domains separately might allow a fully-developed objective approach, but that is not possible here. Our contribution is entirely in the novel use of higher-order programming in GE grammars for aesthetic problems.

This paper is structured as follows. Higher-order programming is desribed in Section II, with a motivating example. Section III describes relevant previous work on abstraction and re-use in the context of aesthetic design problems and general EC. In Sections V and IV we describe the application of higher-order programming applied to two problem domains, music and abstract architectural design. Finally, Section VI draws some domain-independent conclusions.

II. Higher-Order Programming

Higher-order programming involves the use of functions as variable values, as function arguments, and as function return values. It is a powerful method of abstraction, generalisation, and re-use [1], important properties in programming by hand as well as in automatic programming. One main motivation for higher-order programming comes from situations where multiple pieces of code have some behaviour in common but require other parts of behaviour to be specified parametrically, as in the following example adapted from [5] (p. 19).

The authors are with the Natural Computing Research & Applications group, Department of Computer Science and Informatics, UCD CASL, University College Dublin, Dublin 4, Ireland. Contact author's email: jamesmichaelmcdermott@gmail.com. James McDermott is funded by the Irish Research Council for Science, Engineering and Technology under the Empower scheme. Thanks are due to the anonymous reviewers for very constructive comments and suggestions.

[1]The provenance of this quotation is unknown, and it has been attributed to various artists including Laurie Anderson, Frank Zappa, and Elvis Costello. Most are musicians and none are architects.

978-1-4244-8126-2/10/$26.00 ©2010 IEEE

Suppose we have written a function `list_files(dir)` which walks the filesystem's directory structure recursively (starting at `dir`), printing out each filename as it is encountered. Suppose we then need to write a function `print_files` which again walks the directory structure, but this time prints the contents of each file instead of the filename. Of course we can copy and paste our old code, but this is inelegant and violates the "don't repeat yourself" principle: a bug-fix to the original function will not automatically be propagated to the copy-pasted one. It would be possible to put the new functionality into the existing function, and parameterise the behaviour by adding a boolean argument `print_filename_only`. However, this approach will not scale if we then need functions to search each file for a particular string, or to print each file's modification time, or new functions which have not been thought of yet. Instead, it is better to abstract out the common portion of behaviour to a function `walk_directories` which takes an extra argument, a function `fn`. `walk_directories` is therefore a **higher-order function**: it invokes `fn` for each file encountered during the directory walk. We can now achieve maximal re-use of our code by calling `walk_directories(dir, print_name)`, `walk_directories(dir, print_contents)`, `walk_directories(dir, print_modification_time)`, etc. Without higher-order programming, it would not be possible to re-use the directory-walking functionality. Looking at it from the point of view of automatic programming, e.g. GP, we would need to re-evolve the directory-walking functionality every time it is needed: but parameterising it with a higher-order function allows finer-grained re-use.

In this paper, we will use three main aspects of higher-order programming.

- **Functions as arguments** allow us to combine program components in fine-grained ways, as in the motivating example above. Other examples include `map(f, L)`, which returns a new list created by applying `f` to each element of list `L`, and `fmap(Fs, x)` which returns a list created by applying each of the functions in `Fs` to `x`. Note that passing a function as an argument is not the same as passing the *result* of a function. Functions can "cross-cut" each other rather than simply being arranged in hierarchies.
- **Currying** is the process of creating a new function from an existing one by fixing some of its arguments.
- **Lambda expressions** are a literal syntax for the creation of anonymous functions on the fly. Currying is often achieved in this way: for example, we can curry `add(x, y)` to produce a new function of one argument (`y`) as follows: `add4 = lambda y: add(4, y)`. Currying and lambda expressions would be useful in the motivating example above if we wanted a named function for, say, the original `list_files(dir)` behaviour. We can create it by saying `list_files = lambda dir: walk_directories(dir, print_name)`.

Another possibility is passing functions as return values, but this will not be considered in this paper.

The use of higher-order functions implicitly requires the use of a non-trivial *type system*. That is, we must ensure that the values being passed around within our generated programs are of the correct types for their contexts. Types include primitives such as boolean, integer and floating-point, compound types like lists, and functions of various types. In standard GP, all functions' arguments and return values are of a single type, typically either boolean or floating-point. This *closure* property means that any function-call can be substituted for any other. Standard GP thus uses a "trivial" type system. In GP with higher-order functions, we may have (say) functions which return integers and functions which return functions which return integers. The interpreter or compiler will give an error if these are not distinguished. Yu [2] develops a type system sufficient to the task, and the strongly-typed GP approach [6] would also work. An appealing alternative, used here, is GE [7], [8]. In GE, a context-free grammar in BNF (Backus-Naur form) specifies the form program strings may take. It allows the expression of both syntactic (type) constraints and domain-specific semantic constraints.

III. Previous Work

The context for our work is the requirement for methods of *re-use* in EC. Although relatively simple problems can usually be solved with sufficient evolutionary effort (large population size and numbers of generations), for more complex problems which exhibit structure or patterning of any type it is better to use representations which allow re-use of partial solutions (see e.g. [9]). Aesthetic domains typically require such structure. The simplest and best-known approach to re-use is automatically-defined functions (ADFs) as used in GP [9]. ADFs have been used in the context of grammatical evolution (GE) [10], [11], [12]. Similar approaches to modularity and re-use have been successfully used in other representations also [13], [14]. However, some alternative approaches have greater abstraction power than ADFs and similar modular approaches. As shown in the previous section, there are situations where standard functions can not re-use common code, but higher-order functions can.

Meta-programming and higher-level programming in general have been used rarely in EC, with a few exceptions. A *macro* is a function-like construct associated particularly with Lisp and unavailable in most programming languages. Rather than evaluating its arguments and then using their values, a macro leaves its arguments uninterpreted, then writes a new piece of code using them. This allows it to emulate higher-order functionality. Automatically-defined macros [15] work in the context of standard Lisp-based GP, and are very powerful (though this power has perhaps not been fully exploited). The PushGP system, which uses an unusual language and encoding, is also capable of higher-order behaviour [16]. Both of these possibilities are powerful but relatively unintuitive: by contrast, we see higher-order functions as easy to use. They are available in a mainstream

language (Python) with bindings to 3D and music software. Writing a grammar to produce Python code is also easy.

The only explicit use of a standard higher-order programming approach in EC is that by Yu [2]. Yu showed that higher-order functions can lead to better and more generalisable solutions. One mechanism by which this occurs is the creation of behaviours applicable to an input list of any size. This allowed Yu to solve the *general* even-n parity problem, a problem which is difficult without methods of re-use for $n > 4$ and is already difficult using standard ADFs for $n > 6$ [9] (Chapter 6). Hierarchical ADFs improve the situation but not to the point that cases above $n = 11$, or the general case, can be considered.

Yu found that using higher-order functions can have the effect of increasing the proportion of useful programs in the search space, making search more likely to succeed. Random search then performs surprisingly well. If the solution is *compressible*, in the sense that there are repeated behaviours which can be abstracted, then solutions may be much shorter. If good short solutions exist, then the exponential increase in search space size associated with searching for longer solutions can be partly avoided, again making search more tractable. Highly-compressed solutions may also be more fragile, however. Yu's ideas have not previously been extended to the GE representation, nor used in the context of aesthetic EC where patterning of outputs is essential.

A great deal of recent work has focussed on generative systems and open-ended encodings, and increasingly this is seen as the best route to the creation of the staggering organised complexity found in nature [17], [18], [19]. A chief characteristic of such representations is that phenotypes are "larger" than genotypes, but are organised or patterned in some way. In this context, a very useful perspective is given by Woodward [20]. The idea here is that if an optimal genotype is patterned, it presents an opportunity for a method of abstraction and re-use to produce the required pattern at the phenotype level without requiring patterning at the genotype level: "Shortest solutions [ie genotypes] have no structure, if they did any repeated structure could be removed by replacing the repeated structure by appropriate modules." [20]. As a result, the genotype required to represent a particular structured phenotype will be shorter than would have been possible without any method of re-use. With re-use, it is possible for an unstructured genotype to give rise to a structured phenotype. Note that Woodward assumes that the modules available in the system are capable of capturing *any* type of structure. Again, standard ADFs are not capable of this. To take advantage of some types of structure in the problem, stronger forms of re-use are required.

The representation used throughout this paper is GE [7], [8]. Genotypes are variable-length linear arrays of integers which specify the production rules chosen during derivation from a context-free grammar. An intermediate phenotype phase is the string resulting from this derivation. This is interpreted as a computer program which produces an output such as a 3D object or a piece of music. GE is a good representation for our ideas, since it is easy to define fixed or variable numbers of ADFs and HO-ADFs with different numbers and types of arguments.

Another grammatical formalism which has been widely and successfully used in generative art and music is *Lindenmeyer systems* (L-systems) [21]. L-systems have in common with GE that they work by string re-writing. Because of the parallel expansion of all non-terminals simultaneously in L-systems, they typically give structured results. This is an advantage they share with our ADF/HO-ADF approach. By contrast with L-systems, the HO-ADF approach has been used very rarely in EC and never, to our knowledge, in aesthetic EC, and so is open for investigation. The L-system approach encourages fractal-style self-similarity at different scales, which we believe is unnecessary in our chosen problem domains, and so is not considered further.

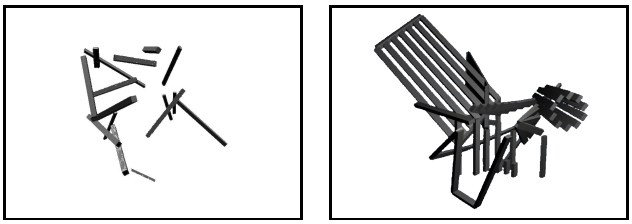
IV. Architectural Design

The application of architectural design is a good medium for the study of abstraction and re-use in aesthetic EC. We will restrict ourselves to a limited sub-domain of architectural design: the space of frame designs, i.e. designs constructed from homogeneous beams of variable length, similar to the trusses discussed by Cagan [22]. This sub-domain, though very limited, is already sufficiently complex to raise difficult representational issues. Shea and Cagan develop a sophisticated representation to produce relatively simple structures. In our previous work [23], we have studied this domain, finding that interactive GE was a good alternative representation, well-suited to representation of the design space and interactive search. However, we also found that a large evolutionary effort was often required to produce interesting designs, and that too many designs suffered from obvious failings, making the user's job tedious and unrewarding. We now report on further work in the same sub-domain, improving on our previous work by analysing the effects of higher-order programming.

A. Separating Behaviour and Data

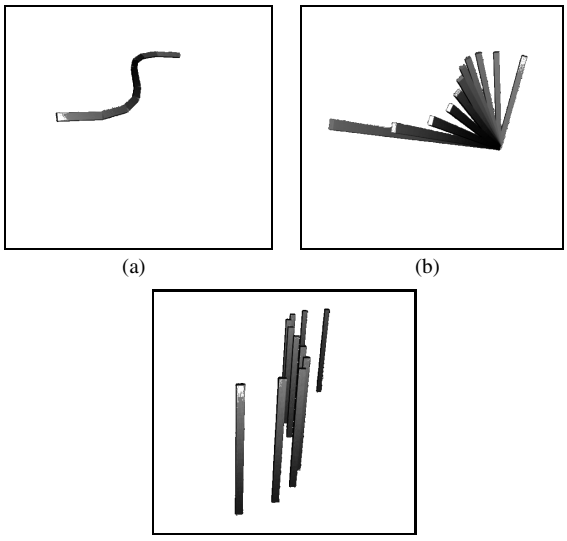
Consider a representation which produces a recursively-grown list of independent beams, with their end-points each represented as (x, y, z) co-ordinates. Such a representation is lacking in methods of abstraction and re-use. It is possible to find *any* object in our design space, but the chances of constructing an object in which all beams are connected according to an overall coherent design are slim: see Figure 1 (a). Let us therefore consider representations allowing abstraction and re-use.

We firstly introduce the idea of geometric *paths* represented by ADFs. A path is a straight line, ellipse, spiral, sinusoidal, or bezier curve. As well as placing beams between neighbouring points along a path, we may wish to use path data in other ways. We can achieve this using higher-order functions. We create a set of primitive (not higher-order) functions which, given a point, produce a beam. Examples include `connectToOrigin` (given any point, create a beam



(a) (b)

Fig. 1. In (a), no re-use. In (b), re-use without higher-order combination.



(a) (b) (c)

Fig. 2. Re-using a bezier curve with higher-order `map`. In (a), `follow(path)`. In (b), `map(connectToOrigin, path)`. In (c), `map(perpendicularZ, path)`. Each of the three uses the same bezier path function, but the *usage* of the path is separated from its definition. Both aspects are available for re-use.

from that point to the origin (0, 0, 0)) and the anonymous function created by currying the `beam` function itself: `lambda pt: beam(pt, <pt>)`. Here, `pt` is an argument to the function, but `<pt>` is a non-terminal which will be given a value during the grammatical derivation. Given any such point-to-beam function, we can use the higher-order function `map` to get an object created by calling that function for *each* point along a path. A path can now be used in multiple ways: this is a type of abstraction. See Figure 2.

We are also using currying and lambda abstraction here: a function `perpendicular(point, plane)` is available, which returns a beam given a point and an integer indicating which plane to drop to (e.g. 2 indicates the $z = 0$ plane). The `perpendicularZ` function, used in Figure 2, might be created by currying: `perpendicularZ = lambda x: perpendicular(x, 2)`. Similar methods are used to create variant `connect` functions. In both of these cases, the aim is to produce a new function which takes a single point as an argument, so that the new function can be used as an argument to `map`. An anonymous curried function such as `lambda x: connect(x, (3, 5, 7))` allows re-use of the point (3, 5, 7) across multiple invocations of the function.

So far we have separated the creation of path data from its usage and obtained a flexible method of re-combining and re-using them. One potential advantage of this scheme is that during interactive evolution, a user might find that (say) a particularly appealing curve is combined with an unappealing variant `connect`, e.g. Figure 2(b). The separation of path data from behaviour means that a single favourable mutation might be enough to produce the same curve combined with (say) `perpendicularZ`, as in Figure 2(c).

We also need the ability to apply multiple behaviours to a single point: this is considered next.

B. Iterating over Functions

In order to produce larger and more complex designs, it is possible to create multiple paths and apply a behaviour to each. However, we again have the problem that the multiple paths may be entirely unsuitable for each other. For example, they will in most cases not result in a design which is entirely self-connected: see Figure 1 (b). An alternative strategy is to create a list of functions, and create a higher-order function which returns the result of applying each function to a given point. This is precisely the `fmap` function described in Section II. In the simplest case, this will result in a design where several behaviours meet at a single point. One feature enabled by this idea was requested by architectural students during classroom trials of our software: the ability to use a particular style of joint developed by the students, characterised by the meeting of four beams.

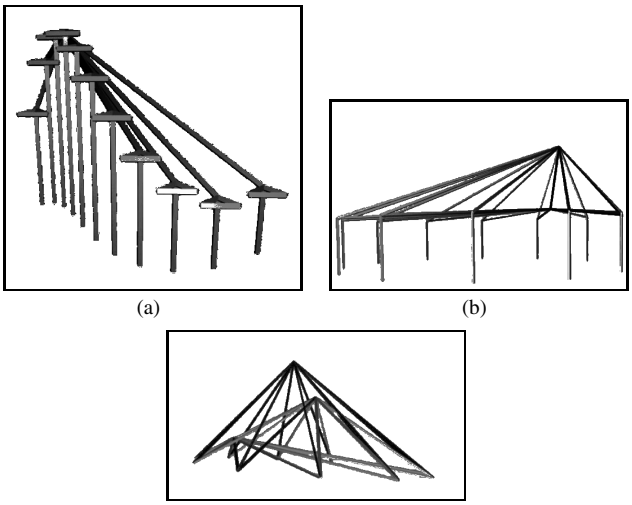
The true power of `fmap` appears when we combine it with `map`, so each point in a list created by `map` is passed successively as the point argument to `fmap`. This combining method increases flexibility greatly, while encouraging the essential properties of connectedness and overall coherence. Three individuals, created by random sampling of a small number of generations of size 15, in three independent runs, are shown in Figure 4 (c-e). A simplified version of the grammar used to produce these individuals is shown in Figure 3. From the point of view of grammar design, the most interesting point is the ease with which the user can change the number and definitions of ADFs (`<scalar_pt_func>` and `<point_shape_func>`). The higher-order function is `<map_fmap>`. No HO-ADFs are used here.

To construct complex paths without making the vast majority of paths appear random, we combine simple paths by addition (see `add` in Figure 3) to produce compound paths. We have also implemented just one of the many possible functions which create a compound object located at a given point: `triangle` (see Figure 4(a)) creates a small triangle oriented in the x, y, or z-plane (in future work we will allow such functions to be evolved rather than hard-coded).

The individual whose code is given in Figure 6 and is pictured in Figure 5(a)) uses one instance each of `map_fmap` and `map` to combine two `triangle` functions, one `perpendicular`, and a path created by adding a linear path and a sinusoidal one. This example demonstrates the idea of "stylistic" as opposed to entropic mutation. The original individual and three new ones created by a single mutation are shown. The mutated individuals retain a strong stylistic

```
# compose list of functions with list of points
<scene> ::= map_fmap(<point_to_shape_funcs>, <points>)

# create organised list of points using map. scalars(n)
# returns a list of n scalars evenly spaced in [0, 1]
<points> ::= map(<scalar_point_func>, scalars(<n>))

# functions which, given a scalar, return a point
<scalar_pt_func> ::= lambda t: diagonal(<pt>, <pt>, t)
                   | lambda t: ellipse(<pt>, <r>, <r>, t)
                   | lambda t: sinusoid(<period>, t)
                   | lambda t: bezier(<pt>, <pt>, <pt>, <pt>, t)
                   | lambda t: spiral(<phase>, <period>,
                                      <r>, <bezier>, t)
                   | lambda t: add(<scalar_pt_func>(t),
                                   <scalar_pt_func>(t))

# functions which return a shape, given a point.
<point_shape_func> ::= lambda pt: beam(<pt>, pt)
                     | lambda pt: perpendicular(pt, <dim>)
                     | lambda pt: triangle(pt, <dim>)
                     | lambda pt: connectToOrigin(pt)

# points are represented as tuples
<pt> ::= (<x>, <x>, <x>)
# <period>, <x>, <dim> are numbers
```

Fig. 3. Simplified version of the grammar used to produce our architectural designs.

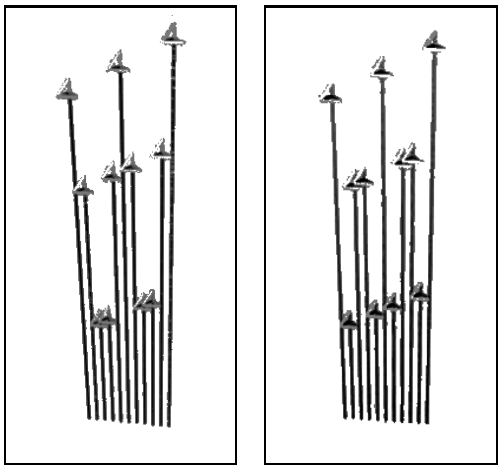


(a) (b) (c)

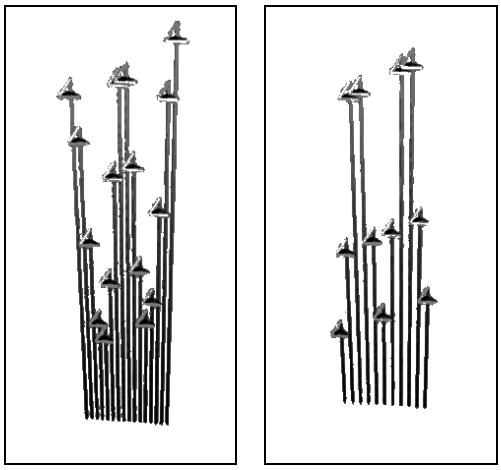
Fig. 4. Combining `map` and `fmap` for coherent, compound designs.

similarity to the original. Many components are changed by the mutation, but in a consistent way.

Crucially, the representation of this individual is quite minimal: little or no further abstraction is possible. This is the end-point of the process implied by Woodward [20]. The underlying genotype has no structure, but the phenotype design is patterned and coherent. Mutations result in interesting variations which are still well-organised: see Figure 5. There are many well-formed and stylistically similar designs in the vicinity of the original individual, and so we claim that this representation is conducive to successful IEC. The search space has been "enriched" with a higher proportion of individuals which are self-consistent in the sense that each component conforms to an overall structure (to see this, imagine offsetting just one component of 5(a) in the y-axis, i.e. "into" or "out of" the page). Even early generations



(a) Original (b) Mutation



(c) Mutation (d) Mutation

Fig. 5. W-shaped gate: each of the three individuals created from the original by a single mutation retain a strong stylistic similarity to the original. The design's organisation is not destroyed, even though many components are changed in some way.

tend to produce results we find subjectively interesting, which helps to alleviate the *fitness evaluation bottleneck* in interactive EC [3], [4].

We have succeeded in our goal of abstraction and reuse. Our designs are quite complex, yet organised; and our grammar is quite open-ended, capable of producing a wide variety of very different styles of structure. Creating similar structures without advanced methods of abstraction and reuse would be possible but would require a huge evolutionary effort to produce an individual composed of many individual components, carefully aligned. By contrast, the individuals shown appeared in the first 3 or 4 generations in a run with population size 15, in which selection was not applied.

V. Music

We introduce our study of higher-order functions in the context of music by working a toy example. Consider the very simple hand-written piece of music shown in Figure 7, which exhibits several types of pattern. How should we represent the musical search space? For the sake of simplicity, in

```
map_fmap([lambda x: triangle(x,0),
          lambda x: perpendicular(x,2),
          lambda x: triangle(x,2)],
         map(lambda t: add(
               lambda t: linear(t, ((14,10,8),
                                    (9,15,12)))(t),
               lambda t: (0.0, 0.0, 11*(
                 1.0+sinusoid(1*4*pi*t)))(t)),
             scalars(10)))
```

Fig. 6. W-shaped gate: original code.

this example, we assume that all notes are of equal duration and volume and that accidentals (sharps and flats) and rests are not allowed.

The simplest possible representation will simply see us store 32 integers in an array. This representation has several obvious disadvantages, as with the individual-beams representation considered at the beginning of Section IV. The vast majority of the search space will be comprised of random-sounding pieces. A large evolutionary effort will be required to produce a coherent-sounding piece. Mutations will be highly entropic—that is, they will gradually degrade a piece, rather than transforming it into a distinct but stylistically similar one. The underlying issue, which is too large to be fully addressed here, is the contrast between stylistic similarity and "Hamming" or "edit-distance" similarity.

We seek a representation which does not suffer these disadvantages. In order to exploit the structure obvious in the piece, we will look for methods of abstraction and re-use. The first compressible feature we notice is the simple ascending 4-note pattern, which is repeated multiple times starting on different notes. Given a function `ascend(start-note, n)`, we can call it with $n = 4$ and different `start-note` parameters to produce each of the instances. The second argument could be curried to produce a new function `ascend4(start-note)`. This avoids the duplication of the 4. Similarly, a `repeat4(start-note)` function can create the two instances of repeated notes. The entire piece can now be represented as `ascend4(10) ascend4(6) ascend4(8) repeat4(10)` (etc.). This can be compressed further. With the creation of a new, higher-order function `f(g, h, start-note)` which calls its first argument three times and its second argument once, each at appropriate start-notes relative to its final argument, the piece will be reduced to `f(ascend4, repeat4, 10) f(ascend4, repeat4, 8)`. This suggests another function, `j(f, g, h, start-note, jump)` which calls `f` twice, once at `start-note` and once at `start-note + jump`, passing `g` and `h` in each time. Our piece is now represented by `j(f, ascend4, repeat4, 10, -2)`, which seems very minimal indeed. Much of the de-duplication work could not be achieved without higher-order programming.

We also claim that this representation reflects the composer's or listener's parsing of the piece. In this simple case, it is easy to see: it would be incorrect to view the second half of the melody as new material rather than a transposition of the first half, for example. Our representation captures these obvious structural relationships. In more complex music, including hand-written pieces as well as that produced by our IEC system (see Section V-A below), the relationships are not so obvious and the parsing of a piece becomes partially subjective [24]. Nevertheless, we claim that "paper-based" musical composition sometimes uses thought processes modelled closely by this type of representation.

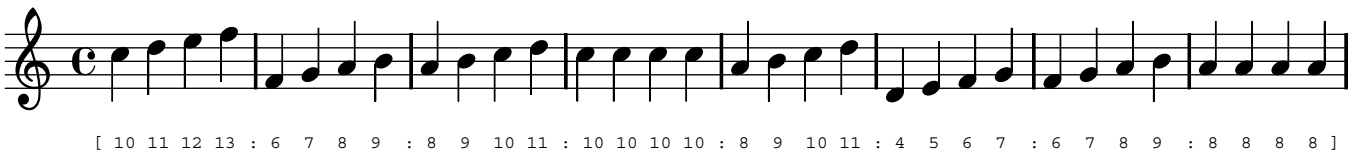
Note that each of the functions given above is to be regarded as an ADF—that is, we are not assuming that such specific functions will be provided for us, perfectly suited to our target piece. Some of them, of course, are higher-order ADFs. Mutations can affect the definition of these functions, as well as the invoking code. It is now instructive to consider the effects of mutations on a representation like this. Changing the 10 to a 12 will transpose the entire piece upwards, rather than putting just some notes out of tune with the rest. Now suppose a mutation occurs in the definition of `f`, so that it follows a (0, -5, -2) pattern instead of the (0, -4, -2) it currently has. Since `f` is re-used, this change will not break the symmetry of the piece. Now imagine that the 4, in the definition of `repeat4`, is replaced by a 3. This change will decrease the piece's symmetry somewhat. It will transform the piece from strict 4/4 to 3/4 with two bars of 4/4, demonstrating that duplication (of the 4) was available for abstraction. Next, these ideas are implemented in a more sophisticated evolutionary/generative music system.

The function `f`, above, essentially "does something n times, then does something else"—where the two somethings are passed-in as functions. Re-use of such a function leads to the reification of this structure as a fully-fledged musical motif—despite the fact that it contains no pitch or time data itself. This type of abstract motif is a perfect fit for higher-order programming by passing-in behaviours. It would be impossible to describe and pre-write all possible functions which might work in this way, since they are so many and varied. Therefore it is necessary to use an ADF-like technique in combination with a meta-programming or higher-order programming technique to evolve them.

A. Music System Implementation

The representation we have chosen is, again, GE with ADFs and HO-ADFs. Here, the GE derivation process produces a program which outputs a piece of music in MIDI format, the final phenotype. The grammar used here is quite complex: a phenotype string consists of some boilerplate code, multiple ADF function definitions, some constants, some variable state, and some invocation code. Although the array representation raises interesting issues, we focus here on the higher-order programming aspects of the grammar, a simplified version of which is shown in Figure 8. From the point of view of grammar design, again, the most interesting point is the ease with which the user may change the number and definitions of both ADFs and HO-ADFs. The complete grammar and working code is available for download: `http://skynet.ie/~jmmcd/software/mtm_demos_cec2010.tgz`.

The system operates in a manner reminiscent of turtle graphics, in that a cursor with state (position and "orienta-



[10 11 12 13 : 6 7 8 9 : 8 9 10 11 : 10 10 10 10 : 8 9 10 11 : 4 5 6 7 : 6 7 8 9 : 8 8 8 8]

Fig. 7. A short, compressible piece of music, together with a simple integer representation of the pitches. This piece is not an output of our IEC system and is shown for explanatory purposes only.

```
# start symbol
<S> ::= <def_adfs><calls>
<def_adfs> ::= <def_hof0><def_hof1><def_adf0><def_adf1>

# definitions of ADFs and HO-ADFs
<def_hof0> ::= def hof0(fn):{<code_in_fn>}
<def_hof1> ::= def hof1(fn):{<code_in_fn>}
<def_adf0> ::= def adf0():{<code_no_adfs>}
<def_adf1> ::= def adf1():{<code_no_adf1>}

# code usable inside a HO-ADF
<code_in_fn> ::= <command_in_fn>
                 | <code_in_fn> <command_in_fn>
<command_in_fn> ::= <command> | fn()

# code usable inside an ADF
<code_no_adfs> ::= <command> | <code_no_adfs> <command>
<code_no_adf1> ::= <command> | <code_no_adf1> <command>
                   | adf0() | <code_no_adf1> adf0()

# invocation of ADFs and HO-ADFs
<calls> ::= <call> | <calls> <call>
<call> ::= <hof_name(<adf_name>)
<hof_name> ::= hof0 | hof1
<adf_name> ::= adf0 | adf1

# primitives create notes and change state
<command> ::= note() | triad() | rest()
              | set_pitch_delta(<int>)
              | set_current_pitch(<int>)
              | set_time_delta(<int>)
```

Fig. 8. Simplified grammar for generation of music. We pass a function `fn` into the HO-ADFs, and it can be called from inside.

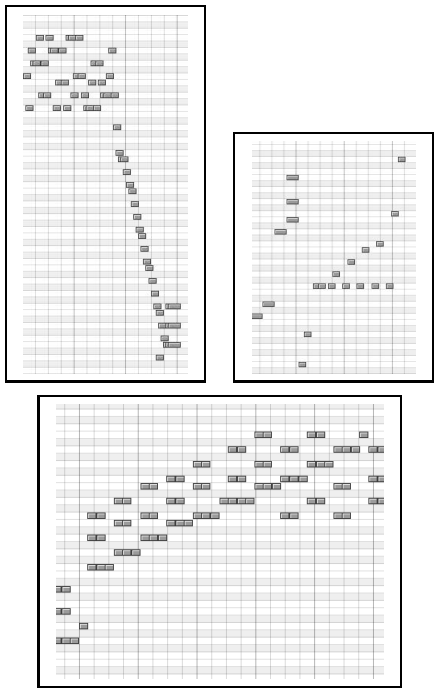


Fig. 9. Score fragments from pieces generated using a higher-order function grammar similar to that of Figure 8.

tion") moves about in a score, sometimes dropping notes or chords at its current position. The basic commands available in the grammar include `note`, `rest`, `set_pitch_delta`, and `set_time_delta`. The ADFs can be passed as arguments to HO-ADFs, allowing complex patterns of re-use. The bodies of both the ADFs and HO-ADFs are created through a recursive grammar technique, where the allowable commands are built up into a list. `adf1` can call `adf0`: though superficially similar, this is not as powerful as allowing a function argument to be passed in, because when the function to be called inside a function's definition is specified as a parameter it allows the HO-ADF to be re-used in multiple ways. This can not be achieved using standard ADFs.

Figure 9 shows some fragments extracted from the scores produced in this way. These pieces have visible structure, and (subjectively) sound as if they have some intention behind them, rather than sounding "random". A simple example is the third piece shown in Figure 9, which appears to be "going somewhere" and then "arrives". But pieces are usually not structured in an overly-repetitive way (the simple worked example in Figure 7 exhibits this problem), because patterns are superimposed with each other or varied slightly. Although the pieces are very short (it might be possible to produce complete pieces by stitching several together as in [25]), they are already interesting to listen to: they can be downloaded from `http://skynet.ie/~jmmcd/software/mtm_demos_cec2010.tgz`. Within a single run it is often possible to spot variations on a common theme, rather than multiple slightly broken versions of a piece—evidence that the genetic operators search at the right level. Crucially, these pieces have been produced with very little evolutionary effort, typically arising in the first 3 or 4 generations with a population of 10. This is important because our central claim is not about the high standard of these pieces, but rather about the encoding's effect on the search space. Related individuals share an abstract, perceptual kind of similarity (even if they are very different as seen by a hamming-style measure), and the space is enriched to the point that early generations are rewarding and encourage further evolution.

VI. Conclusions

In this paper we have introduced higher-order programming and applied it in two aesthetic EC problem domains with interesting results. We have used three techniques char-

acteristic of higher-order programming, *currying*, *anonymous functions*, and *higher-order functions*. In the grammars we have given for both application domains, re-use is the central idea, and HO-ADFs allow types of re-use not possible with standard ADFs. Because we have studied two very different problem domains, we conclude that higher-order programming has four properties of particular benefit to *cross-domain* aesthetic EC:

- **Compression of genotypes**: a piece of music or an architectural design must reach a particular "size" in terms of numbers of components. If a codon was required to place every note or every component precisely, genotypes would need to be large. HO-ADFs allow each function call to produce multiple components and so achieve compression. For example, the original W-shaped gate of Figure 5 uses about 25 codons to produce a design of 33 components (or more if each triangle is counted as three components rather than one), each requiring the specification of two end-points of three dimensions each.
- **Patterning in phenotypes**: our designs and musical outputs are not intended as finished products, but they exhibit very clear patterning or structure even in early generations. This enriches the search space, making the user's job much more pleasant and easier. Figure 4, for example, showed patterned designs created with little or no evolutionary effort.
- **Non-entropic mutations**: starting from a given phenotype, a single mutation does not move a note out-of-place, or move a beam component to a position inconsistent with the established pattern (as might happen with a direct encoding). Such mutations would gradually degrade a good design. Instead, a single mutation can make many small changes which are self-consistent, making a new but still balanced pattern. This was observed in the W-shaped gate example of Section IV and in the initial worked example of Section V.
- **Natural expression of artistic knowledge**: The theme-and-variation pattern, for example, was observed to be expressible with the HO-ADFs of Figure 8.

In both domains, using a few functions which can be combined in multiple ways, we have obtained interesting outputs even in early generations. This goes towards a central issue in interactive EC, the fitness evaluation bottleneck [3], [4]. We believe that higher-order programming is of potential benefit to many aesthetic EC problems.

This paper has made a qualitative argument for the techniques of higher-order programming. We believe that a qualitative approach is useful in aesthetic domains despite the danger of subjectivity. In future work we will attempt to study the same issues quantitatively.

REFERENCES

[1] J. Hughes, "Why functional programming matters," *The computer journal*, vol. 32, no. 2, p. 98, 1989.

[2] T. Yu, "Hierarchical processing for evolving recursive and modular programs using higher-order functions and lambda abstraction," *Genetic Programming and Evolvable Machines*, vol. 2, no. 4, pp. 345–380, 2001.

[3] J. A. Biles, "GenJam: Evolution of a jazz improviser," in *Creative Evolutionary Systems*, P. J. Bentley and D. W. Corne, Eds. Morgan Kaufmann, 2002, pp. 165–187.

[4] H. Takagi, "Interactive evolutionary computation: Fusion of the capabilities of EC optimization and human evaluation," *Proc. of the IEEE*, vol. 89, no. 9, pp. 1275–1296, 2001.

[5] M. J. Dominus, *Higher order PERL: transforming programs with programs*. Morgan Kaufmann, 2005. [Online]. Available: http://hop.perl.plover.com/

[6] D. J. Montana, "Strongly typed genetic programming," *Evolutionary computation*, vol. 3, no. 2, pp. 199–230, 1995.

[7] M. O'Neill and C. Ryan, *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Kluwer Academic Publishers, 2003.

[8] I. Dempsey, M. O'Neill, and A. Brabazon, *Foundations in Grammatical Evolution for Dynamic Environments*. Springer Verlag, 2009.

[9] J. R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, Massachusetts: The MIT Press, 1994.

[10] M. O'Neill and C. Ryan, "Grammar based function definition in grammatical evolution," in *GECCO*, 2000.

[11] E. Hemberg, M. O'Neill, and A. Brabazon, "An investigation into automatically defined function representations in grammatical evolution," in *Mendel 15th International Conference on Soft Computing*, Brno, Czech Republic, 2009.

[12] R. Harper and A. Blair, "Dynamically defined functions in grammatical evolution," in *Proceedings of the 2006 IEEE Congress on Evolutionary Computation*, 2006.

[13] G. S. Hornby, "Measuring, enabling and comparing modularity, regularity and hierarchy in evolutionary design," in *Proceedings of GECCO '05*, 2005.

[14] J. A. Walker and J. F. Miller, "Embedded cartesian genetic programming and the lawnmower and hierarchical-if-and-only-if problems," in *GECCO: proceedings of the 8th annual conference on Genetic and evolutionary computation*. ACM, 2006, p. 918.

[15] L. Spector, "Evolving control structures with automatically defined macros," in *Working Notes of the AAAI Fall Symposium on GP*, 1995, pp. 99–105.

[16] ——, "Autoconstructive evolution: Push, pushGP, and pushpop," in *Proceedings of GECCO*, 2001, pp. 137–146.

[17] P. J. Bentley, "Exploring component-based representations - the secret of creativity by evolution?" in *Proceedings of the Fourth International Conference on Adaptive Computing in Design and Manufacture (ACDM 2000)*, I. C. Parmee, Ed., University of Plymouth, 2000, pp. 161–172.

[18] S. Kumar and P. J. Bentley, "Biologically inspired evolutionary development," in *Proceedings of ICES 2009*, ser. LNCS, no. 2606. Springer, 2003, pp. 57–68.

[19] K. O. Stanley, "Compositional pattern producing networks: A novel abstraction of development," *Genetic Programming and Evolvable Machines*, vol. 8, no. 2, pp. 131–162, 2007.

[20] J. R. Woodward, "Modularity in genetic programming," in *Proceedings of EuroGP*. Springer, 2003, pp. 254–263.

[21] J. McCormack, "Evolutionary L-systems," in *Design by Evolution: Advances in Evolutionary Design*, P. F. Hingston, L. C. Barone, Z. Michalewicz, and D. B. Fogel, Eds. Springer-Verlag, 2008, pp. 169–196.

[22] J. Cagan, "Engineering shape grammars: where we have been and where we are going," in *Formal engineering design synthesis*. Cambridge University Press, 2001, p. 92.

[23] M. O'Neill, J. McDermott, J. M. Swafford, J. Byrne, E. Hemberg, E. Shotton, C. McNally, A. Brabazon, and M. Hemberg, "Evolutionary design using grammatical evolution and shape grammars: Designing a shelter," *International Journal of Design Engineering*, vol. 3, no. 1, 2010.

[24] F. Lerdahl and R. Jackendoff, *A Generative Theory of Tonal Music*. Cambridge, MA: MIT Press, 1983.

[25] P. Dahlstedt, "Sounds unheard of: Evolutionary algorithms as creative tools for the contemporary composer," Ph.D. dissertation, Chalmers University of Technology, 2004.