# A METHOD OF GENERIC PROGRAMMING FOR HIGH PERFORMANCE DSP

*Vesa Norilo*
Centre for Music and Technology,
Sibelius-Academy
Helsinki, Finland
vnorilo@siba.fi

*Mikael Laurson*
Department of Signal Processing and Acoustics,
Aalto University
Espoo, Finland
laurson@siba.fi

## ABSTRACT

This paper presents some key concepts for a new just in time programming language designed for high performance DSP. The language is primarily intended to implement an updated version of PWGLSynth, the synthesis extension to the visual musical programming environment PWGL. However, the system is suitable for use as a backend for any DSP platform. A flow control mechanism based on generic programming, polymorphism and functional programming practices is presented, which we believe is much better suited for visual programming than traditional loop constructs found in textual languages.

## 1. INTRODUCTION

The research project presented in this paper builds on the experience gathered during the development of PWGLSynth, a synthesis library for PWGL. PWGLSynth is an interpreted sublanguage in PWGL, consisting of *boxes*, DSP modules built in C++. A visual graph represents a configuration of such boxes, and the PWGLSynth backend takes care of scheduling and executing the C++ modules in correct order.

The design based on a number of building blocks written in a compiled language being combined in a visual environment is a common one. Perhaps the most widely known platform based on this idea is the Pure Data family[1].

Functional constructs, on the other hand, are often well suited for visual representation[2]. Functional, high level languages are traditionally considered as not being suitable for real time use, but this perception is rapidly changing due to developments in high performance programming languages such as Faust[3].

The rest of the paper is organized as follows. Section 2, *Towards a general language* presents the motivation for pursuing user-defined iterative behavior. Section 3, *Types and polymorphism for flow control*, presents a type system and a method for polymorphic function specialization. Finally, the state and impact of the research are discussed in conclusion.

## 2. TOWARDS A GENERAL LANGUAGE

PWGLSynth has been successfully used in many interesting applications, such as score-driven expressive control of physics based models of acoustic instruments[4]. One of the most useful features of the system is the ability to handle multichannel signals, the vector metaphor[5].

Vectors make it possible for a single box to represent many similar operations on a number of different signals. This can be leveraged in the case of polyphonic instruments, or indeed any scenario where multiple instances of the same algorithm operating on different signals is desirable. The vector metaphor presents capabilities similar to loop constructs in textual programming languages. While useful, the PWGLSynth vector is limited. Only flat 1D vectors are supported.

Initially this sufficed for DSP applications, as the semantic meaning of a vectored signal was simply a multichannel audio signal such as stereo or 5.1. However, as more uses were found for vectors, the limitation became more obvious.

### 2.1. Nested vectors

Consider a polyphonic instrument with a number of voices, each of them stereophonic. The logical representation of such a case would be an n-element vector, where n is the number of voices. Each element would then be a 2-element vector, representing a stereo signal.

This is an example of vector nesting. Such constructs can be used to structure signals and data, and are supported by most general programming languages. In musical programming, SuperCollider[6] is known for excellent use of this idiom.

### 2.2. Looping and flow control

Nested vectors is a simple yet demonstrative case where the behavior desired from vectors is closer to a loop or a flow control structure in a traditional textual language than the plain multichannel audio idiom it was designed for.

To satisfactorily address this kind of case it would make sense to enhance the vector metaphor with features found in loop constructs in many general programming languages. In the best case, it would enable PWGLSynth to do away with the built-in vector concept and give full control of iterative behavior to the patch designer.

However, looping constructs are notoriously hard to express visually. While PWGL is a visualization of an abstract syntax tree and therefore perfectly capable of expressing any construct in the underlying LISP language, LISP loops are awkward and counterintuitive when displayed visually.

Functional programming practices are the ones that often translate best into visual form[2]. Recursion, the standard pattern for iterative functional code, might therefore be a better alternative for a visual environment.

## 3. TYPES AND POLYMORPHISM FOR FLOW CONTROL

The vector metaphor in PWGLSynth was an effective solution for the interpreted case. Interpreted code gains efficiency when the syntactic structure resembles the actual execution path taken by the CPU. This has worked against adoption of principles such as recursion as a loop construct, since efficiently handling this case would greatly complicate the interpreter.

Recent development efforts in the synthesis backend are geared towards just in time compilation[7], allowing more comprehensive transformation of code structure between the visual patch and actual machine code. The compiler can be optimized for cases such as recursion. The visual syntax no longer needs to map closely to a structure efficient to execute in a C++ interpreter. These developments inform examination of a desirable synthesis language in terms of general computer science and theory of programming languages.

### 3.1. Functional design patterns

Let us examine a standard box in PWGLSynth, *add-vector*. This vectored primitive can receive to vectors of numbers, and as expected, computes a result vector as an element-wise sum.

For comparison purposes, the functionality of *add-vector* is shown implemented in C, Listing 1, and an elegant functional language called Haskell[8], Listing 2.

Listing 1: *add-vector* in C language

```
void add_vector(float *a, int a_n,
      float *b, int b_n, float *c)
{
  int c_n = min(a_n, b_n);
  for(int i=0;i<c_n;i++) c[i] = a[i] + b[i];
}
```

The C version has a notable procedural feature: the vector index variable *i*. This variable represents the loop state, and is the most difficult part to express visually.

Listing 2: *add-vector* in Haskell

```
addVector _ [] = []
addVector [] _ = []
addVector (a:as) (b:bs) = (a+b) : addVector as bs
```

The Haskell routine is given three definitions. Let the arguments to the function *addVector* be called a and b. The first two definitions state that if either a or b is an empty vector – or list, in Haskell nomenclature – the result is also an empty vector. The third definition applies for non-empty lists: the result of the function call is the sum of first elements of *a* and *b*, appended by the result of an *addVector* call with parameters *as* and *bs* which are simply *a* and *b*, stripped of their first elements.

Any call to *addVector* will therefore result in a number of successive *addVector* calls until the elements in either *a* or *b* run out. In that case, nothing is appended to the result vector and the call sequence is complete. For further discussion on Haskell, the reader is referred to literature[8].

Nowhere is there a state variable. This is the standard way to define iterative behavior in a functional language, and is expected to map neatly to visual representation.

### 3.2. Enhancing the type system

There is a further great benefit to the Haskell way of depicting *add-vector*. In the C Listing 1, single precision floating point is specified and enforced. The Haskell Listing 2 is *generic*, meaning that any data type capable of being summed by the operator + can be contained in the argument vectors.

PWGLSynth has previously only supported single precision floating point numbers. This has made the type system trivial and easy to manage, as there is only one type – a vector of single precision floating point numbers, with a single element vector begin a common special case.

By adapting the three structural primitives of LISP[9], we can assemble and parse nested vectors of any dimensionality and topology. These primitives are shown in Table 1.

Table 1: *Data structure parsing primitives*

| | |
|---|---|
| pair(x,y) | construct a pair of x and y |
| first(x) | retrieve the first member of pair x |
| rest(x) | retrieve the second member of pair x |

A one-dimensional vector of three floating point numbers, expressed as pairs, would then have the structure shown in Listing 3

Listing 3: Vector of three floating point numbers as pairs

```
(FLOAT and (FLOAT and (FLOAT and NOTHING)))
```

The choice of terms *first* and *rest* reflect the conventional method of traversing a list in a functional manner in languages such as LISP[9] or Haskell[8].

### 3.3. Polymorphic functions and pattern matching

As can be noted in Listing 2, multiple function definitions are given for a single symbol. A function with multiple forms is termed *polymorphic*. It is notable that no branches or logic operators are present in any form of the function; the recursion flow control is handled exclusively by *pattern matching*, selecting the appropriate form of the function based on argument type.

For a statically typed environment like PWGLSynth, this presents an interesting possibility. Complete type inferral for the entire patch can be performed just before starting synthesis. Flow control that depends on nothing but argument type can thus be resolved at compile time, resulting in very efficient code.

#### *3.3.1. Generic programming and polymorphism*

When multiple forms of a function are available, a form must be selected every time a function call is made. The method for this selection is the main novel idea presented in this paper. The mechanism is inspired by *duck typing*, the informal term programmers use to indicate that the type of an object is determined by its capabilities. "If it quacks like a duck, and walks like a duck, it is a duck." Duck typing is possible in many high level scripting languages.

Another source of inspiration is the template metaprogramming in C++. Template functions are generic functions that are specialized at compile time for any given set of arguments passed to the function. In effect, separate versions of the template function are created for each function call and argument type.

This idea of specialization can be further enhancing by allowing polymorphic templates. In effect, all functions in the PWGLSynth backend language are such polymorphs. Whenever a function call is encountered, each form of the function is tried by the compiler in a linear order specified by the programmer. Supplied argument type is fed into the abstract syntax tree of the node, and types for all nodes in the tree are inferred.

Certain nodes only accept arguments of certain types. These include the *first* and *rest* primitives shown in Table 1, which require a pair type. Arithmetical operations will require an atomic number type, and binary operations require compatible types. These specialized nodes, *kernel nodes*, represent the operations from which generic functions are built. If a wrong kind of argument is supplied to a kernel node, specialization fails. Instead of fatal compiler error, only the enclosing function form is rejected.

In this case the caller function will proceed to the next available form of the callee function, attempting to specialize it. If no form of the callee function is suitable, the caller function fails and the error is propagated down one level in the call tree. Therefore, specialization details anywhere in the call tree will control polymorphic function selection all the way down to the root of the tree.

### 3.4. PWGLSynth *add-vector* primitive implementation

Making use of all the features discussed so far, an implementation of the original PWGLSynth *add-vector* box, given here as both pseudocode Listing 4, and as a visual mockup of a PWGL patch, shown in Figure 1. It is instructive to contrast these examples to Listings 1 and 2.

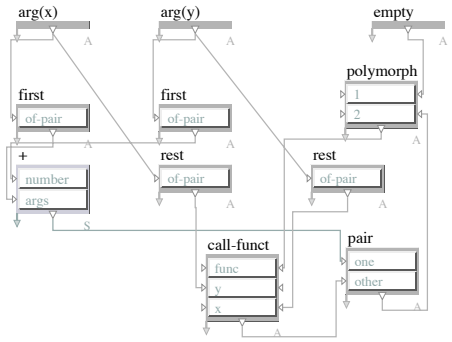


Figure 1: *Visual version of the* add-vector *function*

When presented with the function call, the compiler first tries to apply the latter, more specialized function. It returns the pair built of the sum of the first elements of both arguments and the result of a recursive call for the remaining elements in both arguments.

Once the end of one of the argument vectors is reached, the function fails as the type of argument(x) or argument(y) can not be split. The compiler resorts to the next available form of the function, returning an empty vector.

Listing 4: *add-vector* in the backend language

```
function add-vector = empty
function add-vector = pair(
  first(arg(x)) + first(arg(y)),
  add-vector(rest(arg(x)), rest(arg(y)))
```

#### *3.4.1. Adding support for nested vectors*

It is surprisingly easy to add support for arbitrary dimensions of nested vectors. The pseudocode Listing 5 gives such a function.

Listing 5: *add-vector* with support for nested vectors

```
function flat-first = arg(x)
function flat-first = first(arg(x))

function add-vector = empty
function add-vector =
  flat-first(arg(x)) + flat-first(arg(y))
function add-vector =
  pair(add-vector(first(arg(x)), first(arg(y))),
    add-vector(rest(arg(x)), rest(arg(y))))
```

Two polymorphic functions are specified, *flat-first* and *add-vector*. As long as both arguments x and y are pairs, the *add-vector* function splits the pairs and delegates both halves of the pairs to nested *add-vector* functions, as shown in the third form of the function.

When either *x* or *y* is no longer a pair, indicating end of vector or non-vectored signal, the second form is chosen. This form retrieves the first atom of both arguments, regardless of structure, using the function *flat-first*.

This results in two atoms being added together in the second form of *add-vector*. For the 'empty' or 'nothing' type that signifies the end of a vector, the addition fails, resulting the first form of add-vector being used. It simply returns an empty list, terminating the recursion.

### 3.5. Impact of the type system

As shown above, the new examples show far-reaching definitions of vector behavior expressed within the language. Any user of the system could conceivable redefine such aspects of the system without sacrificing performance.

This level of control is difficult to achieve with a traditional interpreter, as the requisite generality would bring about an unacceptable performance penalty. The problem is resolved here by just in time compilation, which enables more drastic optimization and transformation of the user supplied algorithm.

## 4. CONCLUSION

A type system and a method for polymorphic generic programming was presented, enabling visually logical representation of repetitive routines. The recursive style of functional languages was adapted, and enhanced with a deep polymorphic specialization method. The novelty of the method lies in the powerful abstraction afforded by the axiomatic core language combined with a static structure – yielding a new balance of high level programming concepts and raw computational performance.

The backend compiler is currently in development. The version predating the enhanced type system currently produces very high performance autovectorized code [7]. The new version at this stage produces fast scalar code, but autovectorization is yet to be ported over. The type system has no negative impact on the performance, as all pair handling code will translate to straightforward pointer arithmetic – because of the rigid static typing, all pair structures, no matter how complicated, can be internally represented as flat arrays. The extensive specialization negotiation is carried out entirely at compile time, not affecting run time performance at all.

The compiler is expected to be released as a C-callable library before DaFX 2010, containing facilities for building functions and executing them. The compiler can serve as a high performance computation engine for a variety of repetitive numerical tasks, among them musical DSP.

## 5. ACKNOWLEDGMENTS

This work has been supported by the Academy of Finland (SA 114116 and SA 122815).

## 6. REFERENCES

[1] Miller Puckette, "Pure data," in *Proceedings of the International Computer Music Conference*. International Computer Music Association, 1996, pp. 269–272.

[2] Joel Kelso, *A Visual Programming Environment for Functional Languages*, Ph.D. thesis, Department of Engineering, Murdoch University, 2003.

[3] Yann Orlarey, Albert Gräf, and Stefan Kersten, "Dsp programming with faust, q and supercollider," in *Proceedings of the 4th International Linux Audio Conference*. 2006, pp. 39–47, ZKM.

[4] Mikael Laurson, Vesa Norilo, and Mika Kuuskankare, "PWGLSynth: A Visual Synthesis Language for Virtual Instrument Design and Control," *Computer Music Journal*, vol. 29, no. 3, pp. 29–41, Fall 2005.

[5] Mikael Laurson and Vesa Norilo, "Multichannel Signal Representation in PWGLSynth," in *Conference on Digital Audio Effects*, 2006.

[6] James McCartney, "Continued evolution of the supercollider real time environment," in *Proceedings of the ICMC'98 Conference*, 1998, pp. 133–136.

[7] Vesa Norilo and Mikael Laurson, "Kronos - a vectorizing compiler for music dsp," in *Proceedings of DaFX*, 2009.

[8] Simon Peyton Jones, *Haskell 98 language and libraries: the revised report*, Cambridge University Press, 2003.

[9] Guy L. Steele, *Common LISP : The Language*, Digital Press, 2nd edition, 1990.