A TYPE-BASED APPROACH TO GENERATIVE PARAMETER MAPPING

Josu´e Moreno
Independent Researcher & Composer
Valencia, Spain

Vesa Norilo
Sibelius Academy
Centre for Music & Technology
Helsinki, Finland

ABSTRACT

This paper presents a type-based strategy for automat-
ing parameter mapping. It approaches the problem of in-
tegrating a disparate set of control signals into a musical
application while minimizing the amount of boilerplate
code the user has to write in order to connect control sig-
als to unit generators.

The method is based on polymorphic mapper compo-
nents and signal metadata that allows typed connections.
Concepts for a semantically meaningful parameter map-
ning system for musical applications are explored and im-
plemented in the Kronos programming language.

1. INTRODUCTION

Much of digital instrument building is point-to-point rout-
ing of signals. Often, these signals will need to be adapted
on the go: for an instrument to be playable, its control sur-
face must be mapped so that most of its parameter space is
musically meaningful, maximizing expressiveness while
being as economical as possible.

This results in carefully arranged adaptation layers,
where the output from a control is scaled, translated and
transformed to a range that is appropriate for an audio
generator. Changing either the control or the generator
typically means that the entire adaptation layer must be
redesigned.

Little work has been done to standardize the funda-
mental mapping algorithms [6]. Likewise, few approaches
have been made towards an automatization of mapping,
with the exception of a neural networks based gesture iden-
tification and mapping libraries [8], which are not a gen-
eral purpose solution.

With no standard framework, instrument designers are
constantly re-implementing mapping algorithms whenever
creating a new instrument. Many mapping operations are
very common, so it makes sense to design a general pur-
pose software library that includes these operations [6].

Not all the composers or instrument designers have the
necessary background to envision proper mapping schemes,
resulting in linear and oversimplified parameter transla-
tion. These are rarely musically satisfying.

The authors propose a method to perform automatic
parameter mapping based on an adaptation layer that can
automatically reconfigure itself based on the semantics of
both its input and output.

2. PARAMETER MAPPING STRATEGIES

2.1. Fundamentals of Data Mapping

Data mapping is the process of creating data element map-
pings between two distinct data models. It consists of data
transformation or data mediation between a data source
and a destination.

Data-driven mapping is a recent approach in data map-
ping and involves simultaneously evaluating actual data
values in two data sources using heuristics and statistics
to automatically discover complex mappings between two
data sets.

Semantic Mapping consists of, given a data set, con-
structing a projection matrix that can be used for mapping
data elements from one dimensional space into another,
being or not of the same dimension count. It is an alter-
native to random mapping, principal components analysis
and latent semantic indexing methods [4].

Mapping in musical applications, has been mainly fo-
cused on several strategies formalized as: One to one;
One to many; Many to one; Many to many. These have
been implemented by the composers themselves or within
a software application [10] that may carry aesthetic im-
lications. Also, this categorization is not so much a map-
ning strategy as an overall observation about different in-
put/output configurations one may encounter.

2.2. Survey of the Field

Previously, Steiner has made work towards a Catalog of
Mapping algorithms for the purpose of developing a Map-
ping Framework within Pd. [6] Continuing on previ-
ous developments such as the [hid] library [7], MnM for
Max/MSP [2] and ESCHER [5]. In these frameworks
interaction between mapping modules is based on stan-
dardizing all parameter ranges to a linear range of [0, 1].
Any information concerning the range properties (multi-
dimensional, logarithmic or bipolar) is dropped. Another
approach using normed parameters, in this case of [−1, 1],
is the one used in the UrSound engine [9]. Here, the properties of input and output are taken into consideration after a coercing stage.

An interesting automatization of the mapping process is presented in [8]. The method is based on machine learning via neural networks that can extrapolate a mapping formalism from a data set of inputs and desired outputs. However, this method is mostly geared towards gestures recognition and not intended as a general mapping solution.

These approaches, with the exception of the one presented by Essl [9], require that the user is an expert on mapping strategies. They provide only the fundamental building blocks for the design of such.

Research done on Simplicial Interpolation [1] is of value here as an example of Random Incremental Mapping in the domain of instrument design. These interpolation techniques may produce reasonable intermediate outputs for intermediate inputs out of discretely sampled data. The authors believe that interpolation is not satisfying in a more general purpose mapping engine in which, given the amount of possible scenarios, the nature of the input data and the required output values, might not make sense within an interpolation context.

2.3. Aesthetic Considerations and Goals

Most of the research done in the field of Data Mapping for musical applications carries aesthetic implications. The spectrum of available solutions range from highly developed systems that have a built-in workflow and aesthetic to low level collections of primitives that present a steep learning curve as well as require a plenty of boilerplate programming to achieve simple tasks.

The aim of this research is to automate a lot of the low level programming associated with mapping while remaining as neutral as possible as to how the user wants to map parameters. Type-based metadata is offered as a means for the mapping framework to be smarter, freeing up the user to concentrate on solving the aesthetic problems rather than the technical details.

3. TOWARDS TYPE-BASED MAPPING MONADS

In computer science terminology, parameter mapping can be described as a monad. It is essentially a processing pipeline of functions, each of which transforms the data in some way. In addition, monads can adapt to the context of each sequence step, depending on the function in question and the data being processed. The monad is a very broad concept, but its applicability to mapping is explained in the rest of this section.

3.1. Basic Mapping

Let us consider a simple parameter mapping of a MIDI note to an oscillator, with the addition of a vibrato-producing LFO. A straightforward implementation of this in a low level modular environment might look like the mockup in Figure 1.

In most existing implementations, all the signals consist of plain numerical data. The user is expected to understand that a conversion from a MIDI note number to frequency in hertz is required. The LFO outputs a standard range, typically $[-1,1]$. More domain knowledge is required in order to understand how to scale this into a proper modulation depth; we assume a slider input of $[0,1]$ and treat this as a modulation depth percentage of the fundamental frequency. Finally, the efficient DSP implementation of an oscillator operates in cycles per sample – requiring a final adaptation step from hertz.

The example given is artificially low level to demonstrate the assumptions that tend to be present in even a simple patch. For most adaptation steps, we must actively think about the semantics of both input and output. This knowledge must be reflected in our choice of processing primitives; many conversion primitives are present solely to define the types or units to convert between.

All this knowledge already exists in the patch. A MIDI input, a slider and an LFO have a well-defined output range. Likewise, the oscillator has a well-defined input semantics. Why, then, does the user need to repeat all this by programming each transformation step manually?

3.2. Typed Signals and Polymorphic Primitives

To enable smart behavior for mapping primitives, the authors propose typed signals and polymorphic primitives that generate a mapping monad. Metadata, such as an appropriate unit, parameter range or curvature can be attached to a signal to endow it with desired semantics. This is called type; in contrast to raw data, type carries an annotation of what the data means.

Primitives can then be overloaded so that they respond differently according to the type of the originating signal. Such functions are called polymorphic; a staple of functional and object oriented programming. The distinct implementations of the polymorphic function are called overload.

This allows for a higher level parameter mapping framework where the concepts are nearer to human thought and further from the fundamental mathematical and logical operations carried out by a computer, and the signal processing framework can deduce most low level adaptation steps from the context.

3.3. The Type-Primitive Matrix

Consider a processing chain that performs parameter mapping from a source to a final destination. Each link is a named process with a certain semantic meaning, such as “limit range”, “smooth” or “modulate”.

A framework with $N$ mapping primitives and $M$ signal types can be envisioned as a $N \times M$ matrix where each cell represents the polymorphic overload for a mapping primitive on an input signal of type $y$. Each overload has an output signal of type $z$. Since $z$ can influence overload resolution of the next step in the processing chain, any named process with a certain semantic primitive can exert influence over the entirety of the chain following it.

This has huge potential for automating the mapping process. For example, when a control signal source changes, typically a large part of the following parameter mapping chain must be adjusted. With appropriate type annotations and overloads, this can happen automatically.

3.4. Taming the Combinatorics

The Type-Primitive matrix in Section 3.3 grows as a product of the number of types and primitives in the system. Implementing such a framework can quickly become non-feasible due to the number of overloads required. Several strategies exist to deal with this.

3.4.1. Subtypes

If the signal types can naturally fit a hierarchical category, subtyping can be leveraged. Common traits of several types can be collected into a supertype, and overloads can consider this instead of every subtype separately. Many object oriented systems implement subtypes.

3.4.2. Generics

Generics are the approach chosen by the authors for the implementation of this framework is generic programming, as it is one of the main techniques of the Kronos[3] environment. Similar results could be achieved with the C++ template metaprogrammer or a dynamically typed language such as Javascript.

A generic mapping primitive is one that doesn’t explicitly define a fixed overload for each type, but rather uses pattern matching to perform overload resolution. Such an overload can be selected based upon some salient traits of the incoming type – for example, whether the type supports ordering or a certain subset of arithmetic.

Some domains may even delegate type specialization to their components. For example, a smoothing filter doesn’t necessarily care about the precise type of the signal passing through, as long as sufficient arithmetic for the type is defined in the case, summation and multiplication. For details on this mechanism, the reader is referred to prior work[3].

Figure 1. Low level mock implementation of a MIDI controlled Oscillator with vibrato

4. EXAMPLE IMPLEMENTATION

4.1. About Kronos

For this research, the Kronos programming environment[3] is used to provide the type system, polymorphism and generative capabilities required by the mapping monads. Kronos is a special purpose programming language intended for musical signal processing tasks. It is targeted to musicians and music technologists who want to create customized signal processing solutions. Anything from effect processors to virtual instruments can be built. The Kronos engine offers a powerful functional programming language, yet it can be easily used with a visual, patching-based interface.

4.2. Simple Case

First we consider a simple amplitude control from a variety of sources. To motivate polymorphism, an overloaded amplifier module is demonstrated. Two instances of the amplifier are included in the patch. One of them is controlled by a slider set in decibels, the other by a envelope with a gain coefficient range of $[0,1]$. By using the semantics of amplification instead of a multiplication, the meaning of this simple program becomes obvious; in this context, a level parameter is decibels is easy to understand.

With a standard multiplication, there’s no context from which to deduce how to handle decibels – as multiplying by a decibel quantity has its own, separate meaning. This context is then provided by the end user, by the means of inserting the appropriate mathematics to compute a linear gain coefficient from an incoming decibel before the multiplication. The authors contend that this is something that should be left to software.

4.3. Multidimensional

In this example, disparate real time control signals are combined into a single three-dimensional parameter. Although the example is contrived, it serves to demonstrate some of the automation provided by the framework.

The example is shown in Figure 3. Three inputs are generated by listening to a MIDI continuous controller, analyzing an audio input and receiving an OSC event stream. All these inputs can assign correct type information to
is the one used in the UrSound engine [9]. Here, the prop-
eties of input and output are taken into consideration af-
after a coercing stage.

An interesting automatization of the mapping process is presented in [8]. The method is based on machine
learning via neural networks that can extrapolate a map-
ping formalism from a data set of inputs and desired out-
puts. However, this method is mostly geared towards ges-
ture recognition and not intended as a general mapping
solution. These approaches, with the exception of the one pre-
sented by Issi [9], require that the user is an expert on
mapping strategies. They provide only the fundamental
building blocks for the design of such a solution.

Research done on Simplicial Interpolation [1] is of
value here as an example of Random Incremental Mapping
in the domain of instrument design. These inter-
polation techniques may produce reasonable intermediate
outputs for intermediate inputs out of discretely sampled
data. The authors believe that interpolation is not satisfy-
ing in a more general purpose mapping engine in which,
given the amount of possible scenarios, the nature of the
input data and the required output values, might not make
sense within an interpolation context.

2.3. Aesthetic Considerations and Goals

Most of the research done in the field of Data Mapping for
musical applications carries aesthetic implications.
The spectrum of available solutions range from highly devel-
oped systems that have a built-in workflow and aesthetic
to low level collections of primitives that present a steep
learning curve as well as require a plenty of boilerplate
programming to achieve simple tasks.

The aim of this research is to automate a lot of the
low level programming associated with mapping while re-
main as neutral as possible as to how the user wants to
map parameters. Type-based metadata is offered as a
means for the mapping framework to be smarter, freeing
up the user to concentrate on solving the aesthetic prob-
lems rather than the technical details.

3. TOWARDS TYPE-BASED MAPPING MONADS

In computer science terminology, parameter mapping can be
described as a monad. It is essentially a processing
pipeline of functions, each of which transforms the data in
some way. In addition, monads can adapt to the context of
each sequence step, depending on the function in question
and the data being processed. The monad is a very broad
concept, but its applicability to mapping is explained in
the rest of this section.

3.1. Basic Mapping

Let us consider a simple parameter mapping of a MIDI
note to an oscillator, with the addition of a vibrato-producing
LFO. A straightforward implementation of this in a low
level modular environment might look like the mockup in
Figure 1.

![Figure 1. Low level mock implementation of a MIDI controlled Oscillator with vibrato](image)

In most existing implementations, all the signals con-
sist of plain numerical data. The user is expected to un-
understand that a conversion from a MIDI note number to
frequency in hertz is required. The LFO outputs a stan-
dard range, typically [−1, 1]. More domain knowledge is
required in order to understand how to scale this into a
proper modulation depth – we assume a slider input of
[0, 1] and treat this as a modulation depth percentage of
the fundamental frequency. Finally, the efficient DSP im-
plementation of an oscillator operates in cycles per sample –
requiring a final adaptation step from hertz.

The example given is artificially low level to demon-
strate the assumptions that tend to be present in even a
simple patch. For most adaptation steps, we must actively
think about the semantics of both input and output. This
knowledge must be reflected in our choice of processing
primitives; many conversion primitives are present solely
to define the types or units to convert between.

All this knowledge already exists in the patch. A MIDI
input, a slider and an LFO have a well-defined output range.
Likewise, the oscillator has a well-defined input semantics.
Why, then, does the user need to repeat all this by
programming each transformation step manually?

3.2. Typed Signals and Polymorphic Primitives

To enable smart behavior for mapping primitives, the au-
thors propose typed signals and polymorphic primitives
that generate a mapping monad. Metadata, such as an
appropriate unit, parameter range or curvature can be at-
tached to a signal to endow it with desired semantics. This
is called type; in contrast to raw data, type carries an an-
notation of what the data means.

Primitives can then be overloaded so that they respond
differently according to the type of the incoming signal.
Success functions are called polymorphic; a staple of func-
tional and object oriented programming. The distinct im-
plemetations of the polymorphic function are called over-
loads.

This allows for a higher level parameter mapping frame-
work where the concepts are nearer to human thought and
further from the fundamental mathematical and logical
operations carried out by a computer, and the signal pro-
cessing framework can deduce most low level adaptation
steps from the context.

3.3. The Type-Primitive Matrix

Consider a processing chain that performs parameter map-
ing from a source to a final destination. Each link is a
named process with a certain semantic meaning, such as
“limit range”, “smooth” or “modulate”.

A framework with \( N \) mapping primitives and \( M \) signal
types can be envisioned as a \( N \times M \) matrix where each cell
represents the polymorphic overload for a mapping prim-
itive \( x \) on an input signal of type \( y \). Each overload has
an output signal of type \( z \). Since \( z \) can influence overload
resolution of the next step in the processing chain, any
named process with a certain semantic primitive can exert
influence over the entirety of the chain following it.

This has huge potential for automating the mapping
process. For example, when a control signal source changes,
typically a large part of the following parameter mapping
chain must be adjusted. With appropriate type annotations
and overloads, this can happen automatically.

3.4. Taming the Combinatorics

The Type-Primitive matrix in Section 3.3 grows as a prod-
ct of the number of types and primitives in the system.
Implementing such a framework can quickly become non
feasible due to the number of overloads required. Several
strategies exist to deal with this.

3.4.1. Subtypes

If the signal types can naturally fit a hierarchical category,
subtyping can be leveraged. Common traits of several
types can be collected into a supertype, and overloads can
consider this instead of every subtype separately. Many
object oriented systems implement subtypes.

3.4.2. Generics

Generics are the approach chosen by the authors for the
implementation of this framework is generic programming,
as it is one of the main techniques of the Kronos[3] envi-
ronment. Similar results could be achieved with the C++
template metaprogrammer or a dynamically typed language
such as Javascript.

A generic mapping primitive is one that doesn’t ex-
plicitly define a fixed overload for each type, but rather
uses pattern matching to perform overload resolution. Such
an overload can be selected based upon some salient traits
of the incoming type – for example, whether the type sup-
ports ordering or a certain subset of arithmetic.

Some domains may even delegate type specialization to
their components. For example, a smoothing filter doesn’t
necessarily care about the precise type of the signal pass-
ing through, as long as sufficient arithmetic for the type is
defined in this case, summation and multiplication. For
details on this mechanism, the reader is referred to prior
work[3].

4. EXAMPLE IMPLEMENTATION

4.1. About Kronos

For this research, the Kronos programming environment[3]
is used to provide the type system, polymorphism and
generative capabilities required by the mapping monads.
Kronos is a special purpose programming language in-
tended for musical signal processing tasks. It is targeted
to musicians and music technologists who want to create
customized signal processing solutions. Anything from
effect processors to virtual instruments can be built. The
Kronos engine offers a powerful functional programming
language, yet it can be easily used with a visual, patching-
based interface.

4.2. Simple Case

First we consider a simple amplitude control from a vari-
ety of sources. To motivate polymorphism, an overloaded
amplifier module is demonstrated. Two instances of the
amplifier are included in the patch. One of them is con-
trolled by a slider set in decibels, the other by a envelope
with a gain coefficient range of [0, 1]. By using the seman-
tics of amplification instead of a multiplication, the mean-
ing of this simple program becomes obvious; in this con-
text, a level parameter is decibels is easy to understand.

With a standard multiplication, there’s no context from
which to deduce how to handle decibels – as multiplying
by a decibel quantity has its own, separate meaning. This
context is then provided by the end user, by the means of
inserting the appropriate mathematics to compute a linear
gain coefficient from an incoming decibel before the multi-
plication. The authors contend that this is something that
should be left to software.

4.3. Multidimensional

In this example, disparate real time control signals are
combined into a single three-dimensional parameter. Al-
though the example is contrived, it serves to demonstrate
some of the automation provided by the framework.

The example is shown in Figure 3. Three inputs are
produced by listening to a MIDI continuous controller,
analizing an audio input and receiving an OSC event stream.
All these inputs can assign correct type information to
their output signal, describing the range of the signal. In addition, the Envelope-Follower module receives two time constants annotated as milliseconds.

*Algorithm: Map* is used to transform all three parameters at once, applying a transformation to a linear range of [−1, 1]. Because the x, y, and z components have associated type information, the polymorphic To → Linear function can deduce the appropriate scaling automatically. The transformed values are combined into a single vector coordinate, which is used as a coordinate for a spatial panner. A simple smoothing is applied by a filter – which can maintain the vector coordinate semantics due to the generic approach described in Section 3.4.2. The spatializer can also leverage polymorphism; type context can imply whether the control parameter is given in polar or cartesian form, and if the panning happens in 2 or 3 dimensions.

5. CONCLUSIONS

In this paper, the authors have demonstrated some of the possibilities and advantages of using polymorphism as a mapping strategy based upon signal semantics. Types are used as metadata that aid composed mapping monads to configure themselves to account for disparate parameter ranges.

We have the advantage of the generative nature of Kronos programming language. To expand upon this we apply a type system to mapping. By abstracting the math into software objects, mapping can be approached as a system of higher level human logic.

Upon this, the authors are developing a framework for Generative Music Composition and Performance based on live audio and sensor data input. By making the mapping process smarter and quicker to prototype we can enable another way of interacting and conceiving generative music.

The framework along with the Kronos compiler will be made freely available to interested parties. It is conceived as a Kronos library, but given its extensive OSC and MIDI communication capabilities, interaction with various software packages is feasible. Finished adaptation layers can also be built into object code that can link into any system that supports a C language interface.

Acknowledgements: Josué Moreno’s work has been supported by the Sibelius Academy Development Centre.

6. REFERENCES


CONSTRUCTION OF AN ELECTRONIC TIMBRE DICTIONARY IN C++ FOR MULTIPLE PLATFORMS

Naotoshi Osaka
Hiromichi Tsuda
Yoshihata Yamada
Tokyo Denki University
School of Science and Technology for Future life
Tokyo, Japan
osaka@im.den-dai.ac.jp, (tsuda,yamada@srl.im.den-dai.ac.jp)

ABSTRACT

We have been building an electronic timbre dictionary as a support tool for sound creators, which provides them with sound materials from environmental sound database. Since in the previous system sub-modules were written independently in multiple languages, we restructured and integrated the subsystems in C++ to improve the processing speed and usability. In order to verify these improvements, we ran usability test using the search function of the three types, which resulted in a 96.3% rate of task completion, and a MOS (easiness) of 2.9, which was better than the previous system. These scores are considered to be satisfactory. This paper describes the system configuration as well.

1. INTRODUCTION

We have been constructing an electronic timbre dictionary for the purpose of supporting computer music composition and multimedia content creation.

For computer music composition and other sound media content creation, finding sound material can be the very first phase of the work sequence. However, recording all the materials on one’s own is costly and sometimes impossible.

We have been constructing an electronic timbre dictionary which provides sound material from an environmental sound database for such sound creators, with various search functions [1][2].

Freesound is a well known system of sound data [3]. The electronic timbre dictionary introduced here has a browsing mode as well as sound registration with an upload function for the users. The major features of our system are: a variety of search functions including inverse look up of the sound using onomatopoeia besides forward search and keyword search, and GUI-based sound exploration: a single stream of a sound is defined as a sound object and represented by a sphere in a 3D domain [2]. Acoustic feature types to define the displacement in 3D space are being under study that fits well to our auditory perception.

Another aspect is that the system aims to act as a high-quality sound database, so that the search results can be used in content creation.

We have built a server-client application and a desktop (stand-alone) application. For the former type, the database can be shared publicly or limited to a closed group. The latter type is useful for personal and private use.

The purpose of the system construction so far is to implement each module and examine each function assigned to the module. Therefore each module has been developed in different languages by different developers the - sound browser mode is written in Adobe Flash and the 3D timbre display function is written in Adobe Flash, while the coordinate mapping of sounds are written in C++.

Resultantly, each module behaves rather independently and only the intrinsic function in each module has been tested, and the sequential flow of two related models or operation between two modules have not been sophisticated. Since the first stage of system goal is almost accomplished, we move to the second stage. Our next goal is in specified sound search. These independent functions should be integrated and the relevant data should be available via fast and shared access between modules.

Previous sub-modules were constructed in different languages and some modules ran as a different application. Resultantly, cross-compilation and linking of two different languages were not necessarily possible and sharing parameters and data became complicated. Because of the frequent update of the flash platform or Flex SDK, its portability was low. Moreover, the resultant application is more difficult to maintain and has slower processing speed than one written in a native program language.

Therefore we have implemented the second version to consider strong cooperation among sub-systems. Specifically, we aimed to achieve three goals: multi-platform (Windows, Mac OS and Linux), fast processing speed, and easiness of linking modules. For these goals we selected a single lower level language: C++.

The system adopts the Creative Commons licence [4] for sound data, which is considered to be suitable for sound content creators. For example, it is the license offered by freesound.

This paper describes the system concepts and system configuration as well as evaluation test results for various sound search methods.

2. SYSTEM CONFIGURATION

2.1. OpenGL as 3D graphics API

For 3D timbre display, OpenGL[5] is adopted as a cross-platform 3D Graphics API. OpenGL is a portable programming interface, and does not depend on the OS and hardware types.

Also, we adopted the GLUT (OpenGL Utility Toolkit), which is a class library of C++ designed based on the OpenGL, which reduces system development costs.