Chmusick, una librería para convertir chucK en un lenguaje tipo Algorave

Esteban Btancur
ITM
essteb@gmail.com

ABSTRACT

En este paper se mostrará una breve descripción de los procesos que llevaron al desarrollo de chmusick, como librería de extensión para Chuck, y su posterior implementación en sus dos versiones actuales, OOP (programación orientada a objetos) y una versión mas inspirada en la programación funcional. Además, se explicarán las razones y las necesidades que han contribuido en la evolución de la librería, que ha tenido siempre un objetivo pedagógico y ha tenido como público y usuarios a personas sin conocimientos previos de programación con el fin de que vean en esta librería, en el live coding, y en la música, principalmente el EDM y la pista de baile, una forma divertida pero al mismo tiempo profunda de iniciarse en el mundo de la programación. Mostraremos como la sintaxis de chmusick esta fuertemente influenciada por otros lenguajes y proyectos previos, empezando por Ixilang (Magnusson 2011), Sonic PI (Aaron et al. 2011) y TidalCycles (Mclean 2014). Para terminar se explicará brevemente como la librería fue desarrollada en un contexto colaborativo, y como esta librería a servido para el esparcimiento de la cultura del live coding en Medellín - Colombia.

1 Introducción

El live coding ya no necesita de una introducción, existen en este momento bastantes ensayos, páginas web y blogs dedicados a hablar sobre el tema; para los propósitos de este paper se dirá que el live code o la programación en tiempo real (se usará el término en inglés desde este punto en adelante) es la habilidad de visualizar el código como una entidad que puede ser transformada "on-the-fly" (Wang and Cook 2004), es decir, al vuelo, y que además, cada cambio en el código debe ser percibido instantáneamente, es en este contexto en el que los Lenguajes de Dominio Específico (DSL) se hacen relevantes.

El live coding de música o de visuales es en este momento un área de investigación vasta, y con esto, la popularidad de los lenguajes de programación y los ambientes de desarrollo para hacer live coding han crecido progresivamente, es el caso de: chuck (Wang, Cook, and Salazar 2015), Tidal (Mclean 2014), SuperCollider y acá un desarrollo importante Ixilang (Magnusson 2011) que es un mini lenguaje parasitario enfocado en la construcción de patrones, Sonic Pi (Aaron et all. 2011) y muchos otros lenguajes para end-users que han sido desarrollados para enfrentar los retos que trae consigo el live coding de formas diferentes y presentando soluciones creativas y originales.

Ahora, el live coding puede ser usado de muchas formas, formas computacionales y no computacionales, pero en este paper nos enfocaremos en el uso que se ha dado de esta técnica en el movimiento Algorave, y el live coding para generar música para la pista de baile, centrada en la producción de loops y patrones. Un algorave es un evento donde se hace Electronic Dance Music (EDM) usando algoritmos, y es claro que este propósito ha llevado al desarrollo de herramientas adecuadas a la pista de baile, "ixilang is a primary example, and features a structured code editor which while text-based, supports visual correspondences. Tidal is another, and although its focus is on speed of use rather than ease of learning" (Collins and Mclean 2014)

2 El problema: "en vivo"

Todos los lenguajes de programación tienen sintaxis específicas diseñadas para cumplir con tareas igualmente específicas, y es por esta razón que los lenguajes tradicionales, entienda por tradicionales: c y c++, python, java, haskell, etc. no son buenos a la hora de hacer música (desde cero), simplemente, no fueron diseñados con ese objetivo y su modelo sintáctico no se acopla de forma natural a las abstracciones propias de la música; un algorave se convertiría en un evento extremadamente lento si se intenta escribir desde cero una pieza estilo techno con las clases y objetos puros de los lenguajes tradicionales, en este caso los de paradigma orientado a objetos (OOP), pasaría mucho tiempo antes de que el público pudiera escuchar algo, con Magnusson "There is little fun in watching a stressed programmer designing algorithms for minutes before a simple sine oscillator is applied in the playback of a silly melody".

2.1 Chuck, sus habilidades y sintaxis

Chuck es un DSL escrito en c/c++, diseñado para el prototipo rápido de objetos musicales, "this rapid prototyping mentality has potentially wide ramifications in the way we think about coding audio, in designing/testing software (particular for real-time audio), as well as new paradigms and practices in computer-mediated live performance" (Wang 2008). Decimos objetos dado que el modelo sintáctico de chuck es OOP y por lo tanto la creación de objetos es su principal objetivo. De los lenguajes principales de la familia de los OOP chuck hereda características tales como, el polimorfismo, sobrecarga de funciones, entre otros.

Es posible manipular funciones que permiten ejecutar análisis de varios tipos, por ejemplo, FFT y IFFT en tiempo real y aplicar los resultados a procesos de síntesis, o modelar de forma simple un sintetizador con técnicas de AM o FM, pero tal como pasa con los lenguajes tradicionales, aunque este DSL fue diseñado para el modelado de objetos musicales, la construcción rápida de una clase para manipular loops (concepto que es vital para el EDM) fue dejada en un muy bajo nivel, por lo que el programador necesitará de muchas líneas de código para lograr un loop completo, veamos este ejemplo para observar mejor el código necesario para crear un loop básico de Dance en la sintaxis estándar de Chuck:

```
SndBuf bd => dac;
SndBuf hh => dac;
me.dir() + "bassDrum.wav" => bd.read;
me.dir() + "hitHat.wav" => hh.read;
60/120.0 => float tempo;
while(true){
    0 => bd.pos;
    (tempo/2)::second => now;
    0 => hh.pos;
    (tempo/2)::second => now;
}
```

En este ejemplo pueden observarse dos cosas, la primera es que producir este mismo resultado en un lenguaje tradicional tomaría tal vez tres veces más código, pero esto es lo que lo hace un DSL, elevar las abstracciones a niveles adecuados para el fin que es diseñado; ahora, también es de notar que aunque esté diseñado para hacer objetos musicales, al programador le tomará 10 líneas lograr un ritmo básico de dance, y solo como base rítmica. Note también que es necesario saber el nombre exacto y la ubicación de los archivos.

El problema con este código es que se debe usar un control de tiempo que debe ser calculado cada vez, aunque esta sintaxis específica es usada de forma similar por otro DSL bastante popular: Sonic Pi, pero en el caso del Pi, la línea del "now" es reemplazada por la palabra clave "sleep" (Aaron, Orchard, and Blackwell 2014). Chuck tiene opciones para esta sintaxis y es posible escribir funciones para la ejecución individual o la generación de ritmos complejos, para esto tiene la palabra clave "spork~", este keyword es una "operation on functions (which serves as entry points and bodies of code for the shreds). A shred, much like a thread, is an independent, lightweight process, which operates concurrently and can share data with other shreds" (Wang, Cook, and Salazar 2015).

El siguiente ejemplo mostrará como chuck permite escribir funciones que pueden ser ejecutadas en concurrencia para lograr ritmos complejos, también en la sintaxis propia de Chuck:

```
SndBuf BD => dac;
me.dir() + "bassDrum.wav" => BD.read;
60/120.0 => float tempo;
fun void bd (float division){
    while(true){
        0 => BD.pos
        (tempo/division)::second => now;
    }
}
spork~bd(2);
spork~bd(3);

day => now; //esta linea es para mantener los threads "vivos"
```

Este ejemplo muestra una forma simple de lograr poli-ritmos sincronizados a un tempo específico. Analizaremos un caso sintáctico diferente para integrar todo en una herramienta que pueda cumplir con los requisitos, riesgos y retos que nos propone un Algorave.

3 Caso de estudio: tidalcycles

TidalCycles (Tidal de acá en adelante) es un mini lenguaje embebido en Haskell (un lenguaje que es puramente funcional) como lenguaje base y SuperCollider como motor de audio a través del uso del protocolo OSC y el desarrollo de SuperDirt. Para hacernos a una idea de la importancia actual de este lenguaje, cabe mencionar que durante la celebración de los cinco años de Algorave, 19 de los 48 actos de todo el mundo que se presentaron usaron Tidal como lenguaje base para sus presentaciones y es por esta razón que lo usaremos como referencia.

Hablando de manera muy general, se puede describir Tidal como un lenguaje diseñado para manipular patrones, concepto que es muy importante dentro de la construcción de el EDM, además fue concebido como un sistema de live coding musical y en sus principios de diseño usa la siguiente premisa: “have strong design pressures. They need to be highly expressive, both in terms of tersity, and also in terms of requiring close domain mapping between the code and the music that is being expressed. As music is a time-based art-form, representation of time structures is key.” (Mclean 2014) Para ver claramente es necesario ver algunos ejemplos sintácticos, en este caso mostraremos el código necesario para escribir el mismo patrón que se mostró en chuck.

```
d1 $ sound “bd hh”
```

Esta línea es un patrón construido con muestras diferentes, “bd hh”. En este caso no es necesario conocer el nombre exacto de la muestra, y por configuración general, la ruta de las muestras está preestablecida. El programador debe conocer el nombre de la carpeta que contiene las muestras y con esa información es suficiente para la creación del patrón.

Fácilmente se puede notar que la sintaxis está reducida al patrón y los detalles de bajo nivel están ocultos para el usuario final. Como puede verse en este ejemplo, la sintaxis de Tidal está adaptada a la escritura de patrones-loops que pueden ser fácilmente manipulados con otras funciones, como por ejemplo:

```
d1 $ palindrome $ sound “arpy:1 arpy:2 arpy:3 arpy:4”
```

Palíndrome es una función de transformación que toma como argumento un patrón y devuelve un patrón transformado, en este caso, genera una línea que consta del patrón original mas el mismo patrón invertido añadido al final del original, logrando este nuevo patrón:

```
“arpy:1 arpy:2 arpy:3 arpy:4 arpy:4 arpy:3 arpy:2 arpy:1”
```

Estos ejemplos nos servirán para ilustrar uno de los modelos de Chmusick, el otro como ya se dijo, está basado en la estructura misma de chuck y es su habilidad para ser OOP.

4 Chmusick

Chmusick es actualmente una librería dividida en dos modelos sintácticos diferentes, el original está escrito usando el modelo mismo del lenguaje que pretende extender, es decir, OOP, y como tal es una colección de objetos con características propias a las abstracciones musicales que pueden ser manipulados en tiempo real gracias a la habilidad de chuck de ser cambiado “on-the-fly” y un algoritmo extra implementado para garantizar la sincronización rítmica necesaria en el EDM; este modelo está en la familia de chuck mismo y de otros DSLs como SonicPi.

Usando este modelo sintáctico también se implementó como lenguaje de primera experiencia un mini lenguaje llamado CQenze (Betancur 2016) que hacía uso de la librería Chmusick de forma extensiva pero pasando por un parser y analizador léxico previamente para transformar la sintaxis a un nivel más cercano a Ixilang. El segundo modelo está pensado para ser modelado de forma cercana a los paradigmas funcionales , al menos en la sintaxis, aunque no por esto deja de ser un OOP por lo tanto es un pariente cercano a Ixilang por el tipo secuenciador de pasos que usa y a Tidal ya que usa la manipulación de patrones para lograr cambios en la música. Este nuevo giro se implementó de manera accidental para solucionar un problema en el algoritmo original que creaba una sobrecarga en la memoria RAM al no hacer un uso adecuado del colector de basura. Cabe resaltar que aunque su modelo de construcción es de algún modo cercano al paradigma funcional, en sí mismo sigue siendo un OOP.

Para entender las diferencias veamos algunos ejemplos de código y así mismo ver las diferencias y similitudes con otros DSL, y al mismo tiempo reconocer el proceso de construcción como un trabajo colaborativo donde cada nueva adición y/o corrección al código fuente se realizaba para resolver problemas que alguno de los usuarios había presentado en su trabajo personal con la librería.

Para empezar se mostrarán dos ejemplos de código donde se pueden ver las diferencias entre las dos librerías, así mismo servirán para explicar los detalles dentro de Chmusick.

Chmusick (1v)

```
Drum base => dac;
base.drum([1,0],[0,1]);
```

Chmusick (2v)

```
Chmusick live => dac;

live.file("bd") => Buffer.d1.read;
live.file("hh") => Buffer.d2.read;

live.play(Buffer.d1,live.every(2));
live.play(Buffer.d1,live.rotate(live.every(2)));
```

En esta comparación es posible ver cómo la sintaxis de chuck ha sido elevada a un nivel más alto, en el primer caso dejando los detalles en la librería misma permitiendo que el programador trabaje desde su expresividad, sin preocuparse demasiado por detalles técnicos, el segundo código está en un nivel intermedio, dado que es posible controlar detalles más precisos como la muestra específica que se quiere usar, pero sin ser tan profundo como la sintaxis cruda; el primer modelo esgrime su parecido a los secuenciadores de paso tradicionales, el segundo, el trabajo con funciones, patrones y transformaciones de patrones.

El primer código es una sintaxis donde los arrays que se usan como parámetros funcionan como switches de encendido y apagado en el patrón. En este caso, ambos patrones son simétricos, pero no es mas que una casualidad dado que es posible implementar poli-ritmos y ejecutar patrones de diferentes tamaños:

```
base.drum([1,0],[1,1,1]);
```

Este código por ejemplo permite un ritmo 3 contra 2, haciendo un Mapalé de la costa caribe colombiana.

El segundo ejemplo, como ya se dijo está orientado al uso y trabajo con funciones, y por lo tanto, aunque el resultado que produce es el mismo, abre la posibilidad de introducir de forma más profunda conceptos computacionales diferentes complementando así las carencias de la primera versión de la librería. Acá se muestra la función every() que permite la creación del mismo array del primer ejemplo, es decir, un array de tamaño 2 donde la primera posición es un uno (1) o encendido y el resto del array ceros (0), y la función rotate() que desplaza o mueve cada posición del array el número de veces que se requieran (por defecto una sola posición pero puede agregarse otro parámetro a la función para especificar el número de posiciones que se quiere desplazar cada posición dentro del array), siendo así una función sobrecargada, como muchas otras en la librería. De la misma forma es posible ver la función file() y play(), la primera toma un archivo de audio, para cargarlo en un buffer de audio estático (estos buffers están nombrados como d1, d2, d3, etc.) y la segunda es la función que permite combinar generadores de audio con patrones, es decir, el primer patrón que recibe es un buffer ya cargado o un generador estándar de chuck como SinOsc o cualquier otro, usándolo para ejecutar el patrón descrito en el segundo parámetro.

La forma o mejor dicho, el estilo sintáctico usado en la versión 1 diferencia cada objeto del lenguaje como una abstracción de un instrumento musical, es decir, Drum es la clase encargada de manipular samples de tipo percutivo añadiendo además métodos para los refills y otros métodos bastante propios a la manera tradicional de pensar la base rítmica. Así mismo es posible instanciar objetos Armónicos y Melódicos de forma independiente con características que le son propias a las abstracciones musicales representadas en un estilo tradicional:

```
Harmony pad => dac;
pad.SinOsc(["Fm","Fm","Fm","Eb"]);
```

Se puede observar como el objeto Harmony recibe como parámetro un array de acordes escritos en notación clásica, característica que le es única y que no puede ser usada en objetos de la clase Drum por ejemplo. Así mismo, la librería tiene objetos para generar síntesis FM, AM u otros tipos ya clásicos de síntesis digital que pueden ser usados de forma similar a los ejemplos descritos, es decir, con métodos diferenciados según su naturaleza.

Con un enfoque diferente, como ya se dijo, esta versión está orientada a la creación de un solo objeto que desde ese punto en adelante, puede ser usado de forma única para el manejo de samples o UGens, bien sean nativos al lenguaje o desarrollados como nuevas clases, por ejemplo:

Figure 1: *Essteb usando chmusick para sonorizar una pieza visual durante el ICLC'16*

```
Chmusick live => dac;
SinOsc bass => dac;
live.file("bd") => Buffer.d1.read;
live.play(Buffer.d1,live.every(2));
live.play(bass,live.every(2,60));
```

En este ejemplo puede verse que el objeto general de control "Chumusik", una vez instanciado con el nombre "live", es usado para la ejecución de samples de la carpeta "bd" pero al mismo tiempo sirve para la ejecución de un patrón específico usando el UGen SinOsc que es un objeto nativo dentro del lenguaje chuck. Es posible usar este objeto de control único para ejecutar todo tipo de patrones bien sea con samples o con síntesis.

5 Colectivo Algo0ritmos

Algo0Rimtos es un colectivo de la ciudad de Medellín-Colombia con una conformación libre, lo que implica que cualquier persona interesada en aprender sobre código, programación, algoritmos, música electrónica o simple curiosidad es bienvenida y aceptada de forma inmediata. El colectivo nace en el año 2014 dentro de un programa financiado con recursos públicos de la Alcaldía de Medellín junto a otras organizaciones que se sumaron al proyecto: MOVA y el Colaboratorio del Parque Explora. La idea fundacional era simple, tomando el manifiesto del algorave, se pretendía la creación de talleres abiertos a cualquier curioso, y por lo tanto era necesario buscar un lenguaje de programación que permitiera el rápido prototipado de música que de una u otra forma fuera bailable o al menos incitara al movimiento, pero al mismo tiempo el lenguaje debía tener una sintaxis que luego pudiera ser aplicada a otros contextos no necesariamente musicales. Fue bajo esta premisa que decidimos escoger chuck como lenguaje base, lo que trajo consigo los problemas ya expuestos anteriormente. En el proceso de este taller, los usuarios fueron los que con sus preguntas, errores, experimentos y demás, lograron poner a prueba el trabajo de programación de la librería. Cada cambio en la sintaxis o el desarrollo de los métodos y miembros de cada objeto de la librería nació en este contexto, donde el trabajo se hacía de forma práctica y luego se evaluaban en grupo los avances y problemas que se hubieran presentado en la cotidianidad, luego de esto se discutían las posibles soluciones o enfoques que podrían tomarse para enfrentar los problemas para luego ser implementados en la librería.

Uno de los puntos claves y la razón por la cual hay dos ramificaciones de la librería es que para garantizar una mayor velocidad a la hora de escribir hay que sacrificar velocidad en la curva de aprendizaje y viceversa, lo que ponía en una balanza constante el trabajo, dado que el colectivo se movía entre algorave y la pedagogía. Cabe notar que, justamente esta ambigüedad permitió una gran variedad de perfiles en la configuración del grupo dado que era posible introducir todo tipo de nuevos enfoques en la forma de resolver los problemas.

6 Conclusiones

El trabajo llevado a cabo durante este periodo ha mostrado que el desarrollo de herramientas para el aprendizaje de la programación y la música a través del live coding es un caso que debe ser profundizado desde lo pedagógico ya que los alcances y toda la potencia del concepto es capaz de ser llevada hasta extremos aún inexplorados. Así mismo, el trabajo colaborativo bien sea entre pares o entre personas con niveles muy dispares es un pilar fundamental dentro del desarrollo de las artes digitales por su misma esencia y definición, y es algo que puede llevar al desarrollo y evolución de las herramientas existentes.

En un futuro es posible definir un estándar para el trabajo en la librería para hacer un release completamente funcional, con documentación completa y guías de instalación, ejecución y personalización. Así mismo el desarrollo de un programa de actividades para la implementación de clases de música donde la herramienta principal sea el live coding o inversamente, clases de programación usando live coding usando la música como excusa y motivación.

6.1 Agradecimientos

Quisiera agradecer a los Algo0ritmos, a todos los chicos que así sea durante años, o durante 1 hora, han contribuido al desarrollo no solo de la librería, sino de la experiencia personal del autor. En especial a Santi(AM)/Jacob(PM), Andres, Vivi, Danny Zurc y claro, al master Federico Lopez. Al ITM por abrir sus espacios para permitirnos jugar. A Platohedro por estar siempre ahí, Crowley, Juan (mi profesor de Español Callejero) y todos los Noiseros de la casa.

Referencias

Aaron, Samuel, Alan F Blackwell, Richard Hoadley, and Tim Regan. 2011. “A principled approach to developing new languages for live coding.” *Nime*, no. June: 381–86.

Aaron, Samuel, Dominic Orchard, and Alan F Blackwell. 2014. “Temporal semantics for a live coding language.” *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modeling & Design - FARM ’14*, 37–47.

Betancur, Esteban. 2016. “Diseño e implementación de un DSL : CQenze, como lenguaje de primera experiencia para el código en vivo.” *Proceedings Festival Internacional de La Imagen ’15*.

Collins, Nick, and Alex Mclean. 2014. “Algorave: Live Performance of Algorithmic Electronic Dance Music.” *Proceedings NIME*.

Magnusson, Thor. 2011. “ixi lang: a Supercollider Parasite for Live Coding.” *Proceedings of International Computer Music Conference 2011 (ICMC ’11)*, 503–6.

Mclean, Alex. 2014. “Making Programming Languages to Dance to: Live Coding with Tidal.” *FARM*.

Wang, Ge. 2008. “The ChucK Audio Programming Language; A Strongly-Timed on-the-Fly Environ/Mentaly;” PhD thesis, Princeton University. http://www.cs.princeton.edu/{~}gewang/thesis.pdf.

Wang, Ge, and Perry R. Cook. 2004. “On-the-Fly Programming: Using Code as an Expressive Musical Instrument.” *Proceedings NIME*.

Wang, Ge, Perry R. Cook, and Spencer Salazar. 2015. “ChucK: A Strongly Timed Computer Music Language.” *Computer Music Journal* 39 (4): 10–29.