Media Modules: Intermedia Systems in a Pure Functional Paradigm

Mark Santolucito  
Yale University  
mark.santolucito@yale.edu

Donya Quick  
Yale University  
donya.quick@yale.edu

Paul Hudak  
Yale University  
paul.hudak@yale.edu

ABSTRACT

We present a Haskell-based architecture to efficiently and easily build systems utilizing many types of media. Haskell’s pure functional paradigm is an increasingly popular programming style and is well-suited to many media manipulation tasks. However, intermedia systems are difficult to build due to foundational concepts of the pure functional paradigm and vast syntactic differences between many of Haskell’s libraries. We use a new approach termed “media modules” for creating intermedia systems with increased modularity and parallelism. Finally, we demonstrate the use of this architecture in a variety of musically relevant scenarios.

1. INTRODUCTION

In building systems to explore and integrate various types of digital media, the subsystems to process sound, user input, and other media types often become entangled. While media processes should be closely intertwined in concept, in many cases it is beneficial to separate media types at the implementation level. For example, it is not uncommon that the best programming style to write a user interface may be different than the best style to express a music processing algorithm. Unfortunately, some programming languages do not make this separation easy.

Modularity and encapsulation are important principles of programming that are facilitated by many imperative, object-oriented languages. Design patterns such as Model-View-Controller cleanly capture many interactive programs and enhance code reusability in imperative languages such as Java and C++. However, these sorts of object-oriented patterns do not fit well into more pure functional languages like Haskell due to the extreme differences in language paradigms.

As a pure functional language, Haskell[1] excels at representing problems in ways that closely mirror their underlying mathematical structures, often allowing much more concise representations than would be possible in other languages. This concision directly translates to development speed and system stability[2]. The language’s pure functional nature also predisposes it to using flat data parallelism, enabling maximum parallelization, and ensuring deadlock-free execution of a task with little manual modification.

However, Haskell has historically found difficulty in the interactive domain. Recent work in the field of Functional Reactive Programming[3] has begun to ameliorate this issue, but it is not without lingering problems. In particular, our work addresses the issue of building intermedia systems in Haskell.

There are many libraries in Haskell for various types of media manipulation, but they do not all work together. This is because many Haskell libraries can be thought of as domain specific embedded languages, or DSELs. Haskell DSELs can be exceedingly different in syntax and style to both other libraries and to the underlying Haskell programming language. This leads to problems when trying to use multiple DSELs in conjunction with each other, as the type systems and syntax can be so discrepant that there is no simple way to integrate the two. Because of this, it is common for users to become “trapped” in a particular DSEL once committing to using it. This can make the design and implementation of intermedia systems a slow and challenging process when features are needed from more than one DSEL, something a modern language should not require of programmers.

We present an architecture using media modules to achieve flexibility through abstraction and clarity through isolation that overcomes these problems in Haskell. Each media module can be independently designed with regard to only an abstraction (a data specification) of the other modules, which both increases code reusability and overall modularity of the program. In some respects, this is similar to the way that abstract classes are used in an object-oriented setting, allowing some parallels to commonly used design patterns for interactive systems. With this approach, simple and effective parallelism becomes obvious and easy to implement, which has fundamental benefits beyond performance.

For music processing, we provide examples utilizing a recently released Haskell library called Euterpea[4], which belongs to the Functional Reactive Programming paradigm. By using media modules, we then integrate this library with two graphical libraries, Elerea[5] and UISF[6]. Our architecture makes these libraries easy to use together, a task that is otherwise laborious and error-prone.

Although our work is specific to functional programming, it addresses general problems familiar to both the computer music and programming languages communities. It is similar to the system in Max[7] and PureData[8] that is used to call foreign functions in languages like C and SuperCollider[9]. Max

Copyright: ©2015 Mark Santolucito et al. This is an open-access article distributed under the terms of the Creative Commons Attribution License 3.0 Unported, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.
has a feature called a subpatch for running user-defined code, and multiple subpatches can be run in parallel. This has the same high-level structure as our media modules. Although common in imperative languages, the approach is uncommon in Haskell. Our system allows for interoperability of multiple Haskell DSELs, placing each in its own media module.

2. ARCHITECTURE

Given the apparent benefits of having well-defined and separated modules, we now present a framework for such a system in Haskell that allows the integration of multiple DSELs. Each media module is constructed using a single DSEL, but without detailed knowledge of other DSELs involved. By design, these media modules control the communication between the various DSELs involved in the overall program, ensuring that each module can run on its own without becoming fundamentally tied to other parts of the program. We connect media modules by using a data channel, an interface that facilitates this limited, but well-defined interaction between modules.

2.1 Media Modules

Every media module will have a somewhat similar structure to many well-known software architectural patterns like Model-View-Controller or Presentation-Abstraction-Control. Generally, these architectures have a user interface, a computational layer, and some mediator between the two. Our design follows a similar pattern, specializing it for use in a media art system. Each media module has the following basic components:

Model: a data structure that is updated on each cycle at the appropriate media rate (audio, video, etc.). This model could be a game state representation, coordinates, pitch information from MIDI messages, etc.

Update Function: a function that modifies the model for a particular part of the program. It will take an old model state and return an updated model state. The rate that this function runs will define the rate of the application (60 fps, 44.1kHz, etc.). For example, a user interface media module might need to alter information in the model as the user changes settings.

I/O Operations: various side effects based on the current model. For example, a MIDI-based program might have an IO manager for receiving input from a keyboard controller and/or sending MIDI messages to a synthesizer.

In an intermedia system, each media type can have a different module. In fact, it is even possible to have multiple modules for each media type.

2.2 Data Channels

A data channel handles communication between media modules. In general, for efficiency, a data channel will carry only a very small amount of data—the smallest amount needed to fit the particular application. For example, we implemented a Kinect controlled pitch modulator where the audio module only needs the body position, not the full Kinect camera data. Depending on the nature of the algorithm(s) reading from the data channel, the channel may only need to store the current values for a particular model or some data may need to be buffered. This situation is illustrated in Figure 1.

The data channel will have a type that all connected modules need to agree upon. If the update functions of two modules operate on the same type of data, those modules can be connected via a data channel. This simplicity of connecting multiple modules is what allows for an elegant solution to building multiple media typed systems.

![Figure 1](image-url)

Figure 1. An example application for a Kinect controlled pitch modulator. The direction of the arrows indicates which modules read from and write to the data channel.

3. IMPLEMENTATION

In pure functional programming, there are no variables. Once a value is created, it cannot be changed. In Haskell, values are updated by writing to new memory locations and using a garbage collector to clean up unreachable entries when they are no longer needed. The lack of variables forces programmers to code in a recursive style, which eliminates the possibility of many bugs experienced by imperative programmers. Because of this, Haskell’s type checker actually goes a long way to ensuring program correctness.

Unfortunately, these aspects of the language can also hinder the effective creation of intermedia systems in Haskell. Our notion of a data channel involves a slight deviation from typical Haskell programming. Fundamentally, we actually need something that behaves like a more imperative-style variable in order to create an area of shared memory between threads. While Haskell does not allow variables in the traditional sense, the Haskell library, STM, allows specially controlled entities called transactional variables [10]. Unfortunately, these transactional variables can be somewhat tricky to manage directly in an unstructured program. Our concept of a data channel encapsulates this functionality to allow users the power of a transactional variable to communicate between media modules. The user will still write in a purely functional style and does not need to worry about low-level details.

In our implementation, a media module is represented as a data structure that includes a polymorphic update function, the type of which determines the type of the data channel, as well as some other information, such as a stopping condition.
to signal that execution should terminate. Two modules are then connected and run in separate threads by opening a data channel, which communicates information between modules using a transactional variable. In Haskell pseudocode, this has the following general format for a two-module system:

```haskell
data MediaModule =
  Module UpdateFun StopCond ...

openChannel module1 module2 = do
  chan <- createChannel
  forkIO (runModule chan module1)
  runModule chan module2

runModule chan (Module u s ...) = do
  x <- readChannel chan
  if (s x) then return () else (u x) >>
  runModule chan (Module u s ...)
```

Consider a simple program for relaying MIDI messages between user-selected input and output devices. The user interface may run smoothly with an update rate of 60 frames per second, but the MIDI back-end should be allowed to run faster to avoid audible timing problems. A media modules approach to this can be outlined as follows and remains within the functional paradigm with no need to consider the threading and data-sharing aspects of the code above.

```haskell
midiFun (inDevices, outDevices) = do
  msgs <- getMidiInput inDevices
  sendMidiOut outDevices msgs

midiMM = Module midiFun ...
guiMM = Module ...
main = openDataChannel midiMM guiMM
```

As shown in the code samples above, the user can write code in a traditional, purely functional Haskell style without regard for the underlying use of imperative-style, mutable variables in the data channel. Our implementation’s source code and further examples of its usage are online at haskell.cs.yale.edu.

4. APPLICATIONS

Media modules have a wide variety of potential areas of application. We briefly outline several of them here.

4.1 Connecting Existing Frameworks

Musical Breakout (Figure 2) is based on an existing implementation of the classic breakout game. The original Haskell version used here was created as an example program for the Haskell DSEL, Elerea[5]. In order to add sound to the game module, we created a data channel to signal when a block was broken. The sound module plays a sound based on which block has been broken. This demonstrates the basic simplicity of this paradigm—no knowledge of the implementation of the game module was needed to connect the sound module.

![Figure 2](image-url) A screenshot of Musical Breakout, which was implemented using two media modules: one for the graphical interface and one for sound. The ball bounces against blocks and the user-controlled paddle at the bottom. When a block is hit and broken, a pitch is played corresponding to the coordinates of the block.

4.2 Interchangeable Interfaces

A step sequencer is a simple musical tool for which one would very reasonably want multiple interface choices. We built example step sequencers using two graphical Haskell DSELS, UISF and Elerea, which differ on a fundamental level. UISF uses a language extension to Haskell called arrows, while Elerea uses a construct called signals. Although these two systems are both in Haskell, they conflict in a way that makes using them together incredibly difficult. Because of this, they would not normally be interchangeable as graphical front-ends for a larger application. Typically, a developer must choose a DSEL at the beginning of coding, and will then have to rewrite a large portion of code if the DSEL needs to be changed.

Using media modules and a data channel, the two DSELS (as interface modules) can both be connected to the same sound module with minimal changes to the code. By specifying the data channel format once, we can easily build more interfaces that are guaranteed to work with the sound module.

4.3 Parallelism

As we develop more complex media systems, we need to be able to leverage modern computer architecture to realize increasingly data intensive creative visions. Haskell, by its pure functional nature, enables straightforward code-level optimization on functions like map[10]. This implementation of media modules is also able to take advantage of parallelism on the module level.

In our implementation, we are able to parallelize programs by giving each media module a separate thread. In general, with an appropriately specified data channel, there is no need to manually synchronize the modules in any way and, therefore, they are free to run at different rates. In discussing clock rates, it is worth noting that a media module is somewhat similar at a high level to what is known as a shred in ChucK[11]. However, in ChucK each shred must be synchronized with regard to a global clock, which presents a set of both performance and design challenges we would like to avoid.
4.4 Educational Use

The media modules paradigm is also useful as a pedagogical tool, as it cleanly separates development of different features of a program. Interactive systems with graphical interfaces are often of great interest to students, but can be complicated to program. For courses that aim to teach programming in a musical setting, it is useful to allow students to modularly address one aspect of a system without having to worry about the others. The media modules approach would allow students to easily replace some parts of a larger application without risking altering the rest.

4.5 Distributed Composition

An application that is the subject of ongoing work is the creation of a distributed algorithmic composition system using media modules. While the examples we have shown so far have been based on a top-down scheme that involves a graphical media module controlling a music-related media module via a unidirectional data channel, it is possible to have bidirectional channels and a collection of interacting modules that do not have a strictly hierarchical control structure. This type of approach would work well to implement interactive performance systems such as those described by Hudak and Berger for modeling jazz [12]. An example of such a system is shown in Figure 3.

Figure 3. An example application of media modules for distributed composition with bi-directional data channels to allow communication between two generative algorithms.

5. CONCLUSION

We have introduced the media module architecture to allow for the design of intermedia systems in Haskell, a pure functional language. We can easily construct parallelized intermedia programs by creating a media module for each media type and a shared data channel for communication between modules. This approach automatically handles otherwise difficult issues such as handling communication between modules despite different updating rates. As the pure functional paradigm becomes increasingly popular, especially in digital media circles, this should be a standard functionality.

But, media modules are more than a technical achievement. Using multiple Haskell DSELs together means not only that the individual programmer has more freedom, but also that the community now has new opportunities to work together. Rather than publishing DSELs that need to compete to be the dominant library, programmers can use whichever style they feel comfortable with, and be assured that their code can be reused by others. Teams can work together on intermedia projects without sacrificing expressivity for compatibility. Intermedia systems are not just a mixing of media, but a mixing of ideas. The programs we write should be a reflection of that ideal.

Acknowledgments

This research was supported in part by NSF grant CCF-0811665.

6. REFERENCES