A modular programming system

Jeff Sullivan

- the motivation -

When I have free time, I tend to spend most of it programming systems that generate signals for human perception – audio, real-time graphics, static images, maybe even physical feedback. In working with systems like this, I've come to realize something: It's not often intuitive to generate signals under an imperative paradigm. If you're trying to synthesize an audio tone in C, for example, you'll probably have a hard time fitting the critical expressions into a single line of code. And it makes perfect sense for that to happen – signals are products of composition, so if you want to create a complex signal, you'll need an equally complex expression. The problem is that imperative languages offer no conveniences when trying to do so.

I wanted to use a language in which a user could construct a system and easily perceive the relationships between its components, even across instances of those components – something more concrete than seeing a variable name inside a function call's braces. This was the first, most important constraint for the language: It should yield visually intuitive systems. Second, since the ultimate purpose would be to create interesting signals, the user should be able to perceive those signals throughout the composition of the system. This introduced two constraints: It needs to be an interpreted language, and it therefore has to be efficient. (After all, it might be generating audio samples at 48 kHz.)

So, I did what I could to balance these goals, and here's what I came up with.

- the overview -

A concise description of what I built is up at the top of the page, there, but let's try to have it actually make sense. I think the best way to describe what I intended to create is actually: a typed component system, a framework for creating component templates within that system, and a program for graphically synthesizing instances of those component templates while enforcing the type constraints.

What is a **component**? Well, it might make more sense to just think of them as functions, with some number of inputs and some number of outputs. I don't just call them functions, though, because they show up in the program as widgets that you hook up to each other, and 'component' seemed more appropriate. As you string together these components, you end up with something that kind of resembles an electrical circuit. Figure 0 shows an example of a 'circuit' that adds together two constant real values.

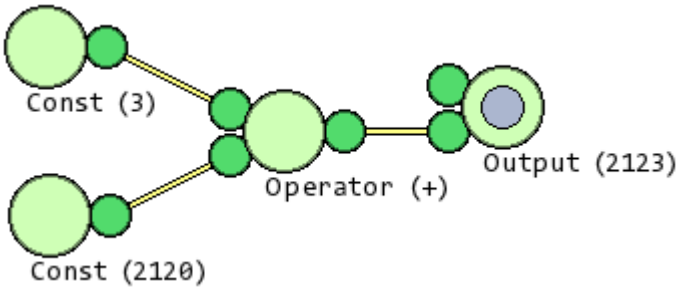


Figure 0: A simple circuit that adds 2120 and 3.

Each large circle represents a component, and the smaller circles that flank them represent their inputs (on the left) and outputs (on the right) – collectively called **patches**. These can be connected, as illustrated by the yellow lines, to pass one component's output to another's input. Hopefully the constant value and operator components are fairly self-explanatory, and as for the 'Output' component, it's just a tool useful for debugging – when you click the blue-gray button, it evaluates its first input. I've so far implemented about a dozen different component templates like these, and the system is designed to make it fairly easy to add more.

Evaluation of a system is so: Starting with some component, such as the 'Output' in figure 0, an evaluation propagates depth-first through the graph, recursively going from each component to all of its inputs. Made linear, an evaluation of the system above might look like this:

```
evaluate Output
    evaluate Operator (+)
        evaluate Const (2120)
            return 2120
        record 2120
        evaluate Const (3)
            return 3
        record 3
        return 2120 + 3
    return 2123
```

As you can see in this example, some patches need not always be filled. (The empty input here can be used to specify how many evaluations the 'Output' component performs when clicked.) However, what you can't see here is how things get interesting as we add more complexity, so let's take a look at all the features that currently exist.

- language features -

- **Arithmetic and Boolean operations**

Nobody wants to use a language that doesn't let you do math and logic. So this language lets you do math and logic. There's a general-purpose binary operator component that provides the four simple arithmetic operations, inequalities, comparisons, and logical conjunctions. There's also a binary 'if' component that branches between two input expressions based on the veracity of its third input (a Boolean expression). As you would expect, this means that expressions in the language are given types – either 'real' or 'Boolean' at this point – and where there are types, there's bound to be...

- **Type checking**

As mentioned before, the program is capable of enforcing constraints imposed by the component templates. Each patch is given a type as part of the component specification, and when interacting with the system, only those patches whose directions (in versus out) differ and whose types agree can be joined by a connection. Rather straightforward for static types, but there's also support for dynamically typed patches, which are defined as follows:

- Dynamically typed patches are initially regarded as having the 'none' type. A patch with the 'none' type can accept a connection of any type except 'none'.
- When a connection from an output with type *T* is made to a dynamic input, that input and all other dynamic patches on the associated component are changed to type *T*.
- When all dynamic inputs have been cleared of connections, all dynamic patches are reset to the 'none' type.

There's a lot going on here, so maybe we can learn better by example again. Consider the slightly more complex, incomplete system in figure 1.

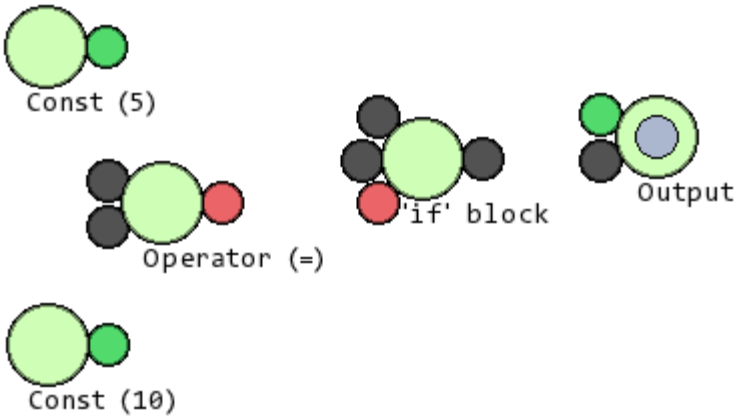


Figure 1: An unconnected set of components.

Here, we can now see patches in a variety of colors, representing their various types. The familiar green is still around, representing real values. Red has been introduced to represent Boolean values, and the black patches are those that are dynamically typed and currently have the 'none' type. Let's see how things change when we hook everything up.

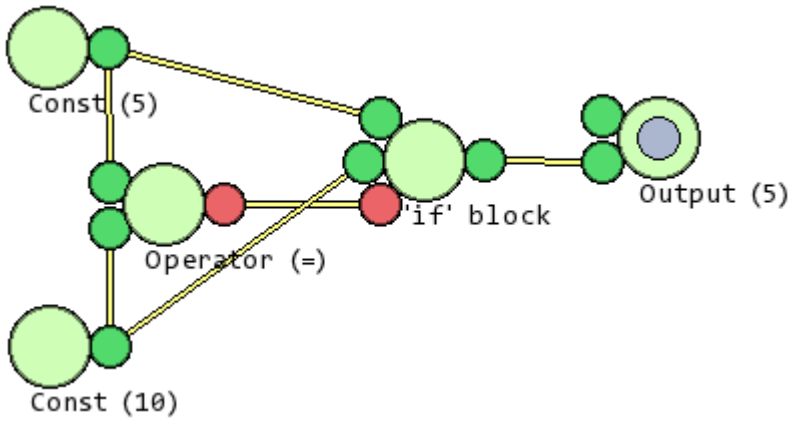


Figure 2: The same components joined into a system.

As you can see in figure 2, the patches have been updated as a result of the connections such that none of them are now given the 'none' type; everywhere you look, we have nice, well-typed expressions. But suppose I had tried to connect the Boolean output of the '=' component to the second input of the 'if' – which, by the way, is the expression that is evaluated in the 'true' case. Even though that input patch is dynamically typed, the connection would have been rejected because the 'false' case input has already been filled by a real-typed 'Const', making all other dynamic patches also real-typed. (I swear this is far more intuitive when you just play around with it.)

You might also notice that the 'Const' components are sending their outputs to multiple input patches. That's something you can do, and it works just as you're probably expecting it to.

- **Lambda expressions**

Ah, yes. This is the big one: The language also implements a lambda calculus, allowing the user to implement functions, which may be recursive, and evaluate them multiple times with varying inputs. The approach I took in designing the implementation reflects my interest in keeping things both 'clean' and efficient – specifically, I wanted to permit recursion without the need for constructing a fixed-point combinator, and I wanted all operations involved in performing an application to execute in constant time. The resultant components warrant a bit of explanation.

The **application** component (called an 'App') has two inputs, and they interact. The second input is an expression that must resolve to a value, and the first input must be a **bound**

expression, which is to say, it must at some point contain a lambda component. When an 'App' is evaluated, it first obtains the value of its second input expression. It then pushes an **application record** to a global stack, recording the value it just obtained as well as a reference to itself, so whoever makes use of the value knows who provided it. Finally, it evaluates its first input.

The **lambda** component creates a bound expression. There isn't really a notion of free variables in this language, because variables are only created by lambdas, and the lambda that creates a variable also binds it. There are two outputs from a lambda – the first simply outputs the value of the variable associated with it, while the second is used in order to perform an application. When the second output is ordered to evaluate, it assumes that there is at least one unused application operation in the evaluation history, which implies that there is an unused variable matching this lambda's type on the global stack. So, the lambda pops an application record from the global application stack, pushes its current value to its own history, pushes a reference of itself to the history of the application component that made the global record, replaces its own value with the applied value, and propagates the evaluation to its input patch.

That's a load of gibberish! You might think so, but these steps are what permit recursion within lambda expressions. The part I left out is that, after the 'App' evaluates its input expression, it pops the most recent lambda from its history stack and 'unapplies' it – which means the lambda pops the top value of *its* history stack and overwrites its variable's current value with that. There are a lot of stacks pushing and popping around here, but let's just say that it works out rather nicely for the user, if not for the implementer.

For example, by using a global stack for applications, the language works as expected even when a lambda and the application that provides its value are vastly separated in the expression graph, so long as they come in the right order. And the way things are implemented, it naturally gives way to structures analogous to **closures** in other languages.

As usual, pictures will probably explain things much better here. Consider figure 3! It implements a function that evaluates the sum of the first n integers, where n is the input.

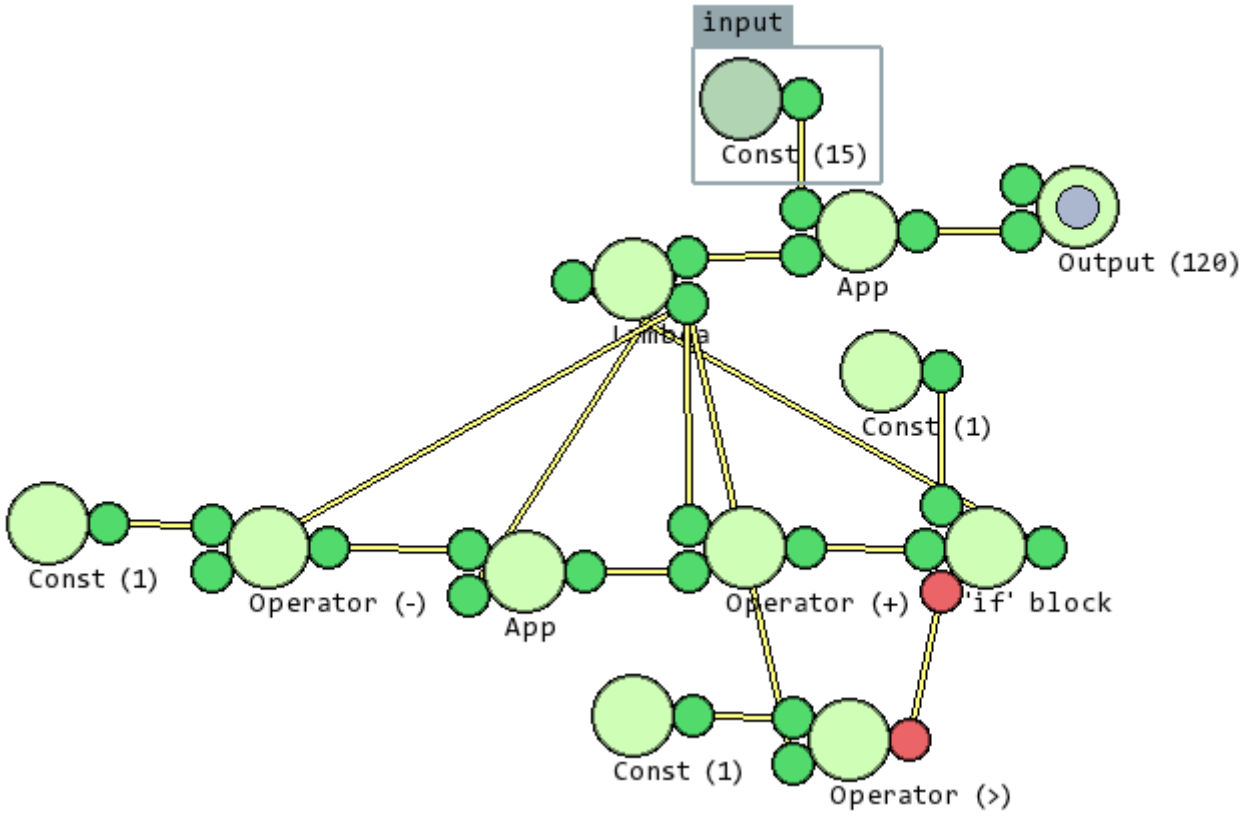


Figure 3: A recursive expression that calculates the sum of the first *n* integers, where *n* is the input.

This system also demonstrates the **group** feature, which will be explained in the next section. Unfortunately, when things start to get recursive, they also get a good deal less intuitive. I didn't quite have time to get around to making the connection lines render more neatly than simple straight lines, so you can easily find yourself in a rat's nest. If you look closely, you might see that the lambda has one connection from its second output to the application's first input, while its first output is used multiple times within its own subtree. If we wish to use the summation function multiple times within one system, we could have the lambda's second output go to multiple application components, each effectively providing a different input value to the function.

As a final note on the topic of lambda expressions, it's also possible to have a lambda that performs exponential recursion, as you might do when implementing a function to find the *n*th value of the Fibonacci sequence.

Now, let's loosen up a bit and talk about things that are just nice to have.

- convenience features -

- **Grouping**

As you saw briefly, the program allows the user to create non-functional structures that house sets of components under a common name, color, and relative positioning. You can use these groups to organize a complex system into subgraphs much like you might have named functions in a verbal language. However, though I had plans to make groups collapsible into single components with equivalent functionality, they really are just something like comments for now. (In fact, you can make an empty group with no size, and just use it as a little comment text box.)

- **Tooltips**

When you hover your mouse over patches, a little tooltip pops up offering a descriptive name, and if the patch is an output, the current value being put out.

- **Serialization**

It wouldn't be terribly productive to use a language that can only exist in an active state. So, I made a save/load feature for the program that writes systems to disk as a list of components and connections. I intended to use this capability for copying/pasting, too, but that's also on the list of not-done-yet features.

And speaking of those, let's speak of them more.

- the agenda -

- **More type checking**

The big thing I had to leave incomplete was the type checking system. Although it does (to the best of my extensively tested knowledge) all that I've claimed it does, there are a few ways I'd like to improve it.

First, it doesn't account for bound expressions generated by lambdas. You can hook up a lambda directly to an 'Output' component and probably crash the program by running it, because the type checker doesn't enforce applications for every lambda expression.

Second, applications aren't smart. The second input to an 'App' (the value it applies to the bound expression) is treated as a special case of a dynamic patch – it's just completely up to the user to make sure the input is the right type. But because of how lambda expressions are

evaluated, it would be entirely possible to have every 'App' deduce what type is necessary for its input to correctly evaluate, and set its input to that type. Or, if an 'App' heads an expression that has branching (perhaps with an 'if' component) and the two branches encounter lambdas with disparate types, the user should be made aware that that's a no-go.

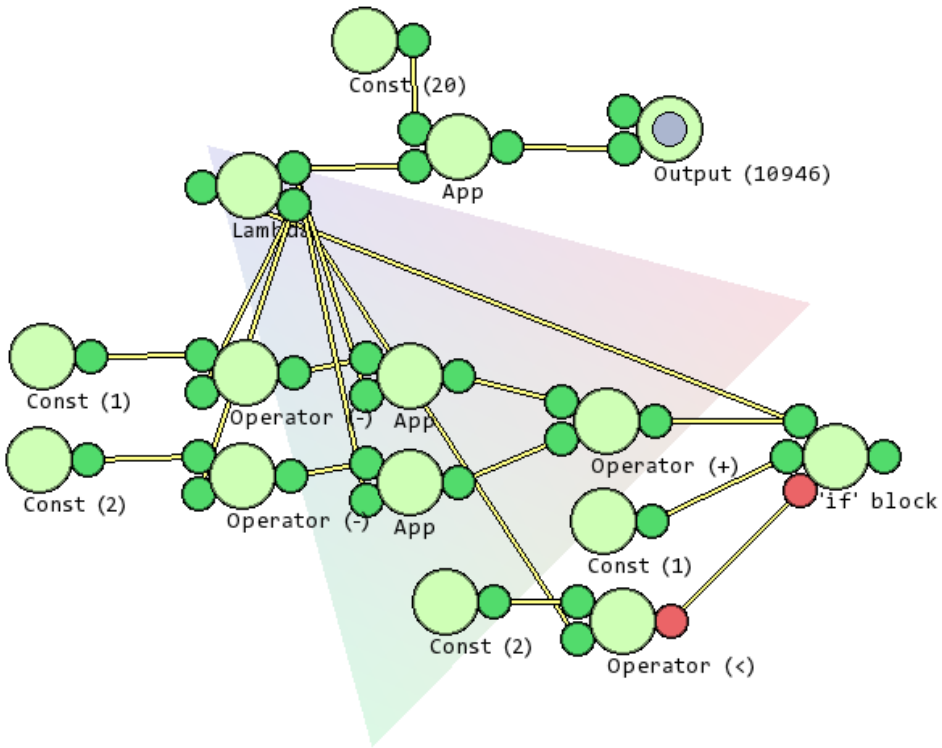
- **Cycle detection**

Or, at least graceful failure. As it stands, evaluation of a cyclical expression that isn't a proper recursion (terminating in a base case) just crashes the program. I certainly wouldn't expect to solve the halting problem, but if an expression contains a cycle and doesn't contain a lambda, it should prevent evaluation.

- final notes -

There's much more to say about what I *could* do with this project, but hopefully the descriptions of what it already does make sense. The figures are all images captured from the program, which I implemented in C++ using wxWidgets for input and event handling, OpenGL for rendering, PortAudio for streaming audio, and FreeType for rendering fonts.

Here's one last figure, showing the sort of Fibonacci function mentioned earlier, along with the yet-unseen, mysterious and powerful triangle:



Have a nice day.