Scritch: A Educational Language for Creating Animations

Jack Barnes and Jack Pattison

June 9, 2021

1 Introduction

1.1 Overview

Scritch is an interactive animation generator that allows users to define objects and transformations which are displayed in an OpenGL render window.

The original goal of the project was to provide an educational functional programming experience, something akin to Scratch or Logo. Scritch is intended for users who are less experienced with programming, especially functional programming. But we do acknowledge that Scritch could also be used by more experienced programmers who enjoy creating graphical animations or interactive games. While the proposed project's goal may have been a far reach for the short development time, we believe that we have made serious progress towards this end. In it's current state, Scritch likely does not meet it's targeted audience of inexperienced programmers, but we believe the building blocks are there to reach this lofty goal.

1.2 Using Scritch

When a user launches Scritch, a browser window is opened, which serves as a graphical user interface for the user. Users can choose the mode (more information below), enter in a program (as a String) in the text area, and run it with a button. Running a program opens an OpenGL render window which displays the result of the input. The user can close the render window, enter a new program or modify the existing program and run the output again.

The current version of Scritch incorporates 2 different modes. The first mode is an "Animation" builder in which the user can define any number of objects and their associated transformations. The user input is parsed into a data type, which is then rendered a looping animation. This mode is the most complete. A simple sample program in our concrete syntax is included in `misc/sample1.txt`. Additional examples are available in the README of the project's GitHub repository.

The second mode is "Play". This mode is less complete, as it does not have a fully-functional parser. It uses an entirely separate set of definitions to define an interactive program. Currently, any program input generates the same example game, a primitive version of Pong.

2 Important types and functions

2.1 Animation mode

The primary building blocks for an animation represented in our abstract syntax are `Transformation` and `Object`.

An `Object` is a record keeping track of an object's name, its position, its direction, and its size, as well as the field `disp`. This last field is mostly vestigial in the project's final state - it was intended to contain information about the shape/appearance of an object, but it is now used to carry information in case of a parsing error.

More interesting is `Transformation`, which represents actions to be applied to an `Object`. We have several primitive `Transformation` constructors for moving an `Object` or changing its size, as well as `Combine`, which turns a list of `Transformation`s into a single one.

Most Transformations are built using `Expr`. `Expr`, our expression data type, carries in its type signature the type which it will evaluate to. Thus a `Pivot` transformation, which changes the direction of an object some amount of radians, requires an `Expr Float` in its construction. The type of an `Expr` is also used in construction of `Expr`s themselves. `Expr` is defined using GADTs, so we can, for instance, enforce that the first argument of `If` is an `Expr Bool`, and its second and third arguments share the same type `Expr a`, where `a` is a type variable.

Constructors for `Expr` which require function arguments use the `Function` datatype, another GADT. `Function` can represent functions of any type or arity.

`Expr` also contains two constructors `Var` and `Let`. Respectively, these represent variable reference and assignment, and require an evaluation function with a notion of state. The monadic evaluation of `Expr`s keeps track of variables using `Map` from `Data.Map.Strict`, mapping `String` variable names to `Dynamic` (from `Data.Dynamic`) values. A simple monadic evaluation function to handle these cases is included in `Anim.Eval`, but this is not currently used by the main program. These two constructions are also not accepted by the parser. Thus variable reference and assignment is currently not allowed in our domain language.

2.2 Parsing

The module `Anim.Parser` implements a monadic parser for the concrete syntax of our domain language (specifically Animation mode). This parser is based on the work of Hutton and Meijer [1]. `Parser` implements the typeclasses `Monad` (and its requirements) and `Alternative`. Because of our use of GADTs in `Expr` and `Function`, type safety is enforced statically and type errors therefore can be detected by the parser.

2.3 Play mode

The top-level type that drives the Play mode rendering is `PlayState` which encapsulates an `ObjMap` (which is a type synonym for a `Data.Map.Strict.Map String Object` and `[EventAction]`.

The `Object` type varies slightly under play mode, defined with a record syntax that maintains, shape, visibility, position, size and direction. Names are not defined in the object, instead being held as the key within `ObjMap`. `Shape` is also a datatype with `Circle` and `Rect Float` as type constructors. Shape determines how the object is drawn and how collision detection determines overlap with other objects.

In play mode, transformations are not their own data type, instead common transformations are of type `Float -> Object -> Object`.

In contrast to the Animation mode's predefined sequential definition for movement, Play mode utilizes an Event driven system. `PlayState` maintains a list of `EventAction`s which are a type synonym for `EventTrigger -> ObjMap -> Float -> ObjMap`. `EventTrigger` is a data type with type constructors `Always`, `GlossKey`, `MouseX`, `MouseY`, and `Collide`. Each time the state is redrawn to the render window, or an input event is generated by Gloss, the events are checked for a matching event which then modifies the `ObjMap` as defined by the `EventAction`. `Always` is a special event which is always run, allowing the state to progress even when no other event is called.

2.4 Gloss

In order to render with Gloss, specific types and functions are required as input. While Gloss is powerful in that it allows drawing OpenGL graphics with very little boilerplate code, it does require very specific input, which drove the direction of much of the development of Scritch.

For the Animation Mode, Gloss requires a function of type `Float -> Picture` which determines the picture to be drawn at that frame. The `Float` value is seconds since the rendering started. For this function `myListAnimator :: [(Object, AnimationSeq)] -> Float -> [Picture]` is used. A single `Picture` can be constructed from a `[Picture]` using a built in Gloss function `pictures`.

For the Play Mode, Gloss requires several more arguments so that it can advance the state, pass events, and finally draw the state. First Gloss needs the starting state, `PlayState`. Gloss will accept any type here, as it is used in following arguments. The next is a draw function for `Playstate -> Picture`, this is implemented with `drawState`.

Now we need to define an event handler `Event -> PlayState -> PlayState`, which is implemented with `eventHandler`. The function `eventHandler` iterates through the list of `EventAction` carried within the state, running each one.

The only `EventActions` that modifiy this state as it passes through are the ones which match the Event that Gloss is passing to the function. This allows a single event to trigger multiple matching events, each modifying the state differently.

Finally a function to advance the state `Float -> PlayState -> PlayState`, we've implemented this is `stepState`. This function works similarly to `eventHandler`, in that it runs through the events, but instead matching on the `Always` Event-Trigger. Additionally, at this point, collision is tested. This detection is fairly basic, accounting for rectangles and circles only. The collision code is within the repository at `src/Play/Collision.hs`, and is likely not worth explaining here.

2.5 WebIDE

The final main component of the project is the user interface, which is generated using the package `threepenny-gui`. How this package works is that it runs a local webserver, which can be accessed from a browser.

Fist, we launch a browser window to the localhost location using the package `open-browser`. Next the GUI started with `runIDE`, which we've given 2 parameters, which are functions that parse a `String` and launch Gloss. The first parameter is for animations, and the second is for play mode. This lets the UI choose which one to use.

To define the WebIDE structure and function, `setup` is defined monadically, returning type `UI ()`. One other interesting note, is that to run Gloss separately from the WebIDE, Gloss (of type `IO ()`) is ran on a separate thread using `Control.Concurrent.runInBoundThread`. Further, to allow the render window to be closed by the user without terminating the entire application, Gloss is built with the GLFW library instead of the default GLUT, using a non-default flag.

3 Design decisions

3.1 GADTS

One significant design choice was the use of GADTs in our representation of the abstract syntax. The highly type-restricted abstract syntax makes writing parsers for the language safer, and fairly mechanical. Most notable is the `Function` datatype, which encodes the type of all functions in our language. In addition to making parsing safer, this type greatly reduces the amount of code needed to evaluate operators - they are all handled by the `op` function, regardless of the number and type of arguments.

The cost of the type-directed parsing enforced by our GADTs is that certain things which could be done with one less safe parser, by ignoring the types of internal expressions, now require multiple parsers. The best example of this is the parsers `iexpr` and `bexpr`. The primary benefit of our parsers is that they allow us to catch a large class of type errors while parsing, without having to separately type-check the program.

The implementation of `evalM` (in `Anim.AST`) demonstrates that this method of type-checking can even be extended, in theory, to expressions containing references to variables. This requires that variables carry type annotations both at assignment and reference.

3.2 Transformation

A very early version of our project saw objects included directly in the expression syntax, as opposed to being used only at the `Transformation` stage. A code snippet of this abstract syntax is included in Fig. 1.

```haskell
module AST where

-- we're just applying functions to arguments
data Stmt = App Function Stmt | Base Basic
    deriving (Eq, Show)

-- some basic data types
data Basic = Obj String
           | Int Int
           | Bool Bool
           | Str String
    deriving (Eq, Show)

data Function = Move Float Float -- Move a number of pixels in x and y directions
    deriving (Eq, Show)

-- TODO figure out what objects are, and what to call them
-- for now we'll just keep track of position and a name
type Object = (String, (Float, Float)) -- placeholder
```

Figure 1: Old abstract syntax

We moved away from this representation for two main reasons. The first was that burying objects deep inside of `Stmt` and nest `Function` applications made it cumbersome to retrieve them when it came time to animate using Gloss.

The second reason for moving `Object` out of what eventually became `Expr`, and then moving `Transformation` as well, was a compromise between the 'functional' goals of our language, and the 'educational' ones. While representing a transformation as a `Function (Object -> Object)` is closer to functional purity, we believed that this could also lead to some unnecessary confusion for our target audience of beginner programmers. The compromise was the current "`Object -> Time -> Transformation`" syntax (where the arrow is part of our concrete syntax, not to be confused with a function arrow), in which programmers specify an object to move, a time for the transformation, and the transformation itself. We believe that this formulation still carries some of the important hallmarks of functional programming, such as creating programs as applications of functions, but it avoids being so dogmatic as to scare newer programmers away.

4 Future work

In conclusion, we have created essentially a proof of concept for Scritch. The code contains all of the building blocks to turn it into a fully fledged language, but these are separated into Play mode, Animation mode, and the not-fully-implemented variable syntax. The only barrier to putting the pieces together is additional time and effort. Another area that could see significant improvement is the concrete syntax. The design of this syntax was not a primary concern in the development of Scritch. The driving idea behind it was to produce something that would be easy to write parsers for. Finally, the UI for Scritch is fairly barebones. When creating a language for beginner programmers, it's important to remember that even something as simple as aesthetics could be enough to put someone off. All of this accounted for, we believe that with some more time to piece together the Animation mode, Play mode, and variable storage, and some more thought put into the language and UI design, Scritch could become something truly special.

References

[1] Graham Hutton and Erik Meijer. Monadic parsing in haskell. *Journal of Functional Programming*, 8(4):437–444, 1998.