Robert Ellwanger (roberte3) 07/24/2024

Combining Functional and Automata Synthesis to Discover Causal Reactive Programs: A Review

Background and Motivation

Program synthesis as a field studies constructing a program which can satisfy a formal specification at a higher level. The application of this is to relieve programmers of the duty to write correct, efficient, backwards compatible code.

Reactive programming focuses on data streams and the propagation of changes. It allows programmers to easily express static and dynamic data streams and automatically propagating changes to the data flow. Below I show an example of the difference between an imperative language and a reactive one:

```
var fizz = 3
var buzz = 5
var fizzbuzz = fizz * buzz
buzz = 2
-> buzz == 15 [ True ]
```

```
var fizz = 3
var buzz = 5
var fizzbuzz = fizz * buzz
buzz = 2
-> buzz == 6 [ True ]
```

As you can see, all changes in the data stream are propagated throughout. An example of this would be FPGA programming (e.g., Verilog).

This paper proposes an idea where reactive programs better mimic the dynamic and reactive reality we live in, which can be automatically synthesized using functional synthesis and automata synthesis.

Functional synthesis is the synthesis of code which does not have a latent state, or a state which is a black box to us, one that cannot be measured or observed. Automata synthesis adds this state.

I found this paper interesting, and was motivated in reviewing it, as I am an automations engineer, and have been working with automation scripting for some time. I was interested in the more innovative details of automation work and tying it in with this degree with machine learning.

Paper Review

The paper is centered around the discussion of the authors' new DSL, a functional reactive language called Autumn, designed after others like Elm.

The novelty in Autumn the authors layout in four parts: Environment setup, Type definitions, Stream definitions, and Event handling.

Throughout the paper, the running example to show the language's niceties is a 2D Mario-inspired platform.

So, to highlight the environment setup, we can use this example, and it will be what defines the play area. It will set up the grid dimensions and background color.

Moving on to type definitions, this is self-explanatory as it defines the object types; however, every object type has a shape represented as a list of 2D positions around the object's center. They also all have a set of internal fields which store additional information (health, stamina, etc.).

```
[Autumn] object Enemy (movingLeft : Bool) ( [(-1, -1, purple), (-1, 0, purple),
                                             (0, -1, purple), (0, 0, purple),
                                           (1, -1, purple), (1, 0, purple)] )
```

```
[Haskell] data Enemy = Enemy
               { movingLeft :: Bool,
                shape :: [(Int, Int, Color)] }
```

Since Autumn, is a **functional** reactive language, you can see some similarities between it and Haskell in syntax and form.

The third part of a program in Autumn consists of Stream definitions, which define object instances and other values and their evolutions over time in a vacuum, void of external events or pressures. This is explained in the paper through the running example of a Mario clone. There would be 4 stream variables in this simplified game: one for Mario, one for the coins, one for the platforms, and a stream variable which represents the hidden latency state of the number of collected and unspent coins. These streams are defined in Autumn using a primitive initnext, which defines a stream of values over time.

[Autumn] *mario = **init** (Mario (Pos 7 15)) **next** (moveDownNoCollision (**prev** mario))*

The variable is initialized with the init expression, and at later timesteps it is evaluated using the next expression. However, you can access the previous value with prev. This is interesting to me as it seems like a two node doubly linked list, but for an object's value rather than for a way to hold multiple objects.

The fourth and final segment of an Autumn program is Event handling, and are expressed using on-clause which means **on event do intervention**:

```
[Autumn] on intersects (prev mario) (prev coins)
           numCoins = (prev numCoins) + 1
           coins = removeObj coins (-> obj (intersects (prev mario) (prev obj)))
```

The event is a predicate which is evaluated and when true, the intersection is one or more assignment, which will override the next value in the stream as defined by the stream types discussed above. As you can see in the provided code, when Mario intersects with a coin, it is removed from the list of available coins, and is added to the number of collected coins.

Another unique aspect of Autumn is that we are allowed to use ***prev*** not only on a stream variable, but also objects within a stream to retrieve their previous value, without resetting all other objects in the stream to their previous values as well.

To acompany this language the authors also introduce AutumnSynth, an end-to-end synthesis algorithm. The input of this algorithm is a trace of a program. In the Mario example used, it would correspond to a sequence of grids for a range of dt time steps, and the user inputs at those steps, with a library of language compnents. With this input, the algorithm produces (or rather synthesizes) an Autumn program in which given the observed sequence of user inputs, it matches the behavior observed in the trace. This algorithm is also broken down into four parts: perception, object tracking, update function synthesis, cause synthesis.

Object perception is the first step and begins by extracting object types and instances from the input sequence of grids. Object instances describe the object's type, position, and any field values. A list of these is extracted for each grid. Object tracking determines how each object changes between the current grid and the next.

Update function synthesis is the next step in the AutumnSynth algorithm. During this step an expression is derived for each object in a grid that describes the change in the object's physical attributes (position and color) in between it and the next grid. The expressions are synthesized using the library of language components passed in.

Cause synthesis is the final step, which searches for an event in an Autumn program that triggers each update function synthesized in the previous step. For this it was assumed that each update function synthesized is unique for that object at that time step. This allowed the researchers to focus on WHY it was triggered instead. This is what this step does. It looks at events in an Autumn program that occur each time an update function was triggered and tries to find one that occurs at exactly those times, every time. If this step cannot find an event trigger for a particular update function, then a latent variable must be added to the program state to now be the event trigger. It does this by searching for the next "closest" event, an event that occurs every time the event does, but there are also times when the event trigger happens without triggering the update function.

The model of the Autumn language consists of:

1. Object types consisting of a shape and list of additional fields [t = < S, data>]
2. States, where each state consists of a tuple which is a set of objects and a set of latent, non measurable, variables [< X_o, X_l >]
3. Actions: key presses, click (containing the 2D position where it occured), no action
4. Observations (the frames or each individual 2D grid)
5. Transition Function: Let a program's history be it's state and it's action H = X x A. The next function defines the modification needed to the current set of variables given a history. The on-clause functions are a set of functions where each function is a tuple < event_i, update_i > where each event is a Boolean predicate, and each update is a function that change the current state given the same input.

6. Observation Function: This takes the shape of an object type and translates it by the position it rendered at on the grid to obtain the observed rendering x_o.

The rest of the paper goes into detail of applying the AutumnSynth algorithm on the Mario clone 2D platformer that was used for an example throughout the earlier parts of the paper. The authors also went ahead and evaluated the algorithm on a custom set of 30 Autumn programs they called the Casual Inductive Synthesis Corpus (CISC). The author's also mention that Autumn language and AutumnSynth were developed in Julia. The interpreter and library functions were an implementation of 1,600 lines of Julia, while the synthesizer was about 18,000 lines.

DISSCUSSION

This paper was very interesting to me after seeing all the hype of "AI is going to take SWE jobs" and products like ChatGPT, and more so the likes of Devin. Having this discussion around the current state of generative code and having computer programs for us. The AutumnSynth feels like a cousin to these LLMs. I was also interested in the approach of a new language that seemed targeted at simple 2D game design, as I have had a large interest in Autonomous Vehicles lately, and found it odd that there really hadn't been a DSL for the automotive industry and autonomous robotics as a whole, I mean they have an entire OS (ROS), but not a DSL to make interacting, controlling, perceiving, etc more intuitive to code. I was also surprised to read that it was only 1.6 KLOCs to write the libraries and compiler, yet 18 KLOCs for the synthesizer (> 10x larger than the language itself!).