StateMap Proposal

Oren Finard *obf2107* - Project Manager

Jackson Foley *jcf2172* - Language Guru

Zuokun Yu *zy2170* - System Architect

Brian Yamamoto *bky2102* - System Integrator

Alexander Peters *arp2169* - Verification and Validation

1 Description

StateMap is a programming language that emulates a DFA (Deterministic Finite Automata). Inspired by the knowledge that a DFA with two stacks can perform any computable operation, the languages purpose is to make the transition from a paper model of a DFA into a program simple to write, and concise in length. The language blends functional and imperative programming styles to allow programmers to abstract away implementations details.

Programmers will code states for their programs through expressions. The only memory that will persist from state to state are those of the Stack primitives, which are declared beforehand. The language has no loops: rather, it has transitions, whereby the code will transition from one state to another, carrying with it only the stack primitives, which act much like LISP lists. The lack of loops limit the code in the states to performing only a constant (upper bounded) number of operations per state, before transitioning to another state (possibly transitioning back to itself: e.g. our languages version of loops) or not transitioning at all (e.g. reaching an end-state). Transitions are evaluated on associated statements: a statement must evaluate to true for a state transition to occur.

The language will ideally be able to run outside programs (i.e. will have a fork/exec ability) so that the language can be used to manage other code, while leaving the implementation details of the various codes up to themselves. We imagine this being used in AI settings (specifically strategy games), though the current vision of the language cannot facilitate inter-program communication (i.e. all inter-program information must run through stdout/stdin).

2 Domain

The mission of StateMap is to provide users with an easy and quick medium to transfer an on-paper DFA representation of an algorithm, problem or game into code. The stack limitation, in addition to a states-only requirement for the code structure, focuses users into maintaining the integrity of the DFA representation.

One of the biggest inspirations for StateMap is an organized and logical implementation of text-based adventure games. The Choose your own adventure genre of books and games involve moving players through various states as they traverse the adventure. Players make decisions, at each state which effects the progress of the game and which states they visit. This type of game is easily implemented using StateMap because the current state of the game can be represented by a State and the decisions the players make are represented by the transitions between States.

Another apparent program type served well by StateMap is decision-making by Artificial Intelligences (AIs). If a game can be easily represented via game-states, then an AIs decision-making would be directly coded as a decision-tree, and the code StateMap would directly reflect the traversal of such a tree.

StateMap is not intended to support the implementation of any programs requiring heavy data-storage and retrieval. Because of the lack of data structures supporting constant time access, StateMap would be inefficient for work involving sorting, indexing, and mapping. This is a design choice, as we wanted to limit the scope of StateMap to a strict DFA representation. These types of algorithms can be done using StateMap, but that is not the intended use of this language.

3 Architecture

The general structure of a program in our language is simply a list of states. States are declared with a name followed by a semi-colon, and all of the logic and transitions that can occur in that state are indented below the declaration. Within a state the programmer can manipulate the stack variables, and define transitions and their conditions. A transition is defined by a condition to be evaluated followed by an arrow indicating which state to transition to if the condition is true. If a state does not contain a transition, the compiler will recognize it as an end state, and the program

will terminate after executing the code contained in it, either returning a value, or void.

There is a single state that is unique to the other states called start. This state is responsible for handling input, initializing the stacks, and finally passing control to the initial state of the DFA.

Global stack variables should be declared at the top of the program, and must have specified types (e.g. int, double, Boolean).

4 An Example Program - GCD

The following is an implementation of the Euclid's GCD algorithm using StateMap:

```
./gcd 9 3

// Declare comments like this.

stack int a;
stack int b;

// initialization
start:
    a.push(9);
    b.push(3);
    *->s1; // * represents anything.

s1:
    a.peek > b.peek -> s2;
    a.peek < b.peek -> s3;
    *->s4;

s2:
    a.put(a.peek - b.peek) -> s1;

s3:
    b.put(b.peek - a.peek) -> s1;

s4:
    print a.peek;
    // No transitions means end state.
```