StateMap

LANGUAGE REFERENCE MANUAL

By

Oren Finard - obf2107
Jackson Foley - jcf2172
Alex Peters - arp2169
Brian Yamamoto - bky2102
Zuokun Yu - zy2170

CONTENTS

1. INTRODUCTION

It has been proven that a PDA with two (or more) stacks can accept any language that a Turing Machine can. From this theorem comes the programming language, StateMap. StateMap is a programming language that is organized and executed in a manner analogous to an Automata diagram, like those seen for DFA's or PDA's. It emphasizes modularity and organization of code into short nodes, which transition to each other until reaching some end state. It shrinks the gap between paper diagram and running code to let the programmer go from algorithmic organization to actual execution quickly and simply.

1.1 DETERMINISTIC FINITE AUTOMATA

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.

1.2 STATEMAP NODES

StateMap programs consist of nodes, and within those nodes there are a constant number of operations, as well as transition statements, which allow for control to permanently leave the current node and execute on a new node. Aside from information stored on globally-scoped stacks, no information is preserved from node to node.

There are two types of nodes: transition nodes, and end nodes. Transition nodes can include transition statements,

which evaluate expressions, and execute if the expression is true. All transition nodes must end with a default, catch-all transition, to ensure that code execution makes its way to an end node. A return node cannot have any transition statements, but it can return data, and control, to the caller. All return nodes must end with a return statement.

Nodes can call sub-automata, which then execute until they reach an end state. Nodes can also make decisions based on the states of sibling automata, which run in parallel to them.

A node within an automata is defined by a name, followed by curly brackets, within which consist of a number of operations (see ‘operations’ section), with either transition or return statements included. There is no keyword needed to define a state as of type ‘end’ or ‘transition’: the language will infer based on whether the last statement in the node is of type transition or return.

1.3 START STATE

A StateMap automata always begins at the ‘start’ state. This necessitates that every automata include a state labeled ‘start’. Automata are organized by declaring the name as “DFA name”, and then within curly brackets defining the rest of the automata.

An automata definition consists of, first, its global stack declarations, followed by an (unordered) list of its nodes, and their definitions. The stacks are typed, and must be declared as such (see code examples).

1.4 MODULARITY

The key to a good StateMap program is extreme modularity. Being able to draw the program, on paper, as an automata means that you are probably on the right track. Nodes are, idealistically, short and concise, and, while the ability to create variables does exist, decisions mostly consist of global information, and local variables exist mostly for convenience, efficiency, and the shortening of code length.

2. LEXICAL CONVENTIONS

2.1 COMMENTS

Both C and C++ style comments are supported.

Multi-line comments begin with characters /* and end with characters */. Any characters may appear inside a multi-line comment except for the string '*/'.

Single line comments begin with the characters // and end with a line terminator.

2.2 IDENTIFIERS (NAMES)

An identifier is a sequence of letters, digits, or underscores, the first of which must be a letter. There is no limit to the length of an identifier.

2.3 KEYWORDS

The following identifiers are keywords and may only be used as such:

return int double float string void DFA main stack char start boolean

2.4 CONSTANTS

There are several types of constants, as follows:

2.4.1 INTEGER CONSTANTS

An integer constant consists of one optional minus sign followed by a sequence of one or more digits. The first digit in an integer constant cannot be a zero, unless it's the only digit.

Valid: 42, 0, -13

Invalid: 042, +13, 00, .25

2.4.2 FLOAT CONSTANTS

A float constant is a 64-bit signed floating point represented with an optional negative, then a significand followed by an e followed by an integer exponent (also optionally signed).

Valid: 2e-13, 1.4, .3e2, 0.0

Invalid: 42, 0

2.4.3 CHARACTER CONSTANTS

Character constants are represented via enclosure with single quotes “’”. No more than two characters can be enclosed.

Valid: ‘a’, ‘\n’, ‘ ‘

Invalid: ‘hello world’, ‘32+1’

2.4.4 DOUBLE CONSTANTS

A double is a float type of 64-bits. It allows for greater precision, and allows for decimal places. The range of values is 15 digits, before and after the decimal point.

Valid: .245, 244.356, -14

Invalid: 2.4e15

2.4.5 BOOLEAN CONSTANTS

A boolean can either be True or False. Any empty value (such as an empty sequence or list) or zero will also evaluate as false. Any other value will be valued as true.

2.5 STRINGS

Strings are represented via enclosure with double quotes ‘”’. To represent the character ‘”’ without closing the string, it must be preceded with a ‘\’. The empty string is represented with ‘””’, with no characters in between the quotes.

Valid: “hello world”, “ “, “32”, “he told me \”yo\””, “”

Invalid: “Clinton said “I did not have””

2.6 PUNCTUATION

2.6.1 BRACES

Braces are used to denote the body of a DFA, or the body of a state in the DFA. The body of a DFA may contain variable declarations and state definitions. The body of a state may contain any number of statements.

2.6.2 PARENTHESIS

An expression may include expressions inside parenthesis. Parentheses can also indicate a function call, or a list of parameters for a state.

2.6.2 SEMICOLON

Used to denote the end of a statement.

2.6.3 COLON

Used to denote a concurrency statement, explained in 3.3.5.

2.6.4 COMMA

Used to separate multiple variable names during type assignment.

Example: String name, address, profession;

2.7 OPERATORS

2.7.1 ARITHMETIC

Operator	Name
+	Addition and String concatenation
-	Subtraction and unary negation
*	Multiplication

/	Division
%	Modulo

2.7.2 ASSIGNMENT

The assignment operator is ‘=’. This assigns the value of the right side of the operator to the left side variable.

2.7.3 COMPARISON

Operator	Name
==	Equality
!=	Inequality
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

2.8 WHITESPACE

Whitespace is defined as the ASCII space, horizontal tab, new-line, carriage return, and comments. Whitespace does not affect the program.

3. SYNTAX NOTATION

3.1 PROGRAM STRUCTURE

Programs are composed of a series of DFAs with a single main DFA. DFA declaration looks like:

```
DFA MAIN(/*ARG1*/ /*ARG2/) {}
```

Additional DFAs without the main identifier follow a similar structure, with the addition of a return type.

```
TYPE DFA NAME( /*ARG1*/ /*ARG2*/) {}
```

3.2 EXPRESSIONS

DFAs are composed of five types of expressions.

3.2.1 DECLARATION AND ASSIGNMENT

{Type}{Id} = {Function} or {Type}{Id} = {Constant}

Note that functions include sub-DFAs. Thus, DFA output may be assigned to variables.

3.2.2 TRANSITIONS

{State}<-* or {State}<-{Literal}{Operator}{Literal} or {State}<-{Method call}

Transition to a state after performing the action on the right hand side of the arrow. The star operator indicates unconditional transition to the state. Since the transitions are evaluated in order, the {State}<-* must be the last transition.

3.2.3 RETURN STATEMENTS

Return {Id} or return {Constant}

3.2.4 METHOD CALLS

{Id}.{Method}({Arguments})

Assume a map called foo was declared. A valid method call is: foo.find(bar)

3.3 STATEMENTS

The types of statements in StateMap are declaration, assignment, function call, transition, concurrency and return. Declaration and assignment are the only two types that can be called outside of a node, i.e. globally in a DFA. Every type of statement must be terminated by a semicolon.

3.3.1 DECLARATION

A declaration statement consists of a variable type followed by an id. Multiple declarations can be made in a single line separated by commas.

```
int i;

Stack<double> s, char c, string s;
```

3.3.2 ASSIGNMENT

An assignment statement is used to set the value of a variable, which can be done during the declaration of a variable, or later using the variable's id. Multiple assignment can be made in a single line separated by commas.

```
int i = 4;

double d = 3.0, char c = 'a', string s = "hello";
```

3.3.3 FUNCTION CALL

A function call statement is a function call expression, but also can be used in an assignment statement taking advantage of the fact that a function call statement has type of the return type of the function.

```
DFA1(arg1);

string s = DFA2(arg2, arg3);
```

3.3.4 TRANSITION

A transition statement consists of a node id, the transition operator and a boolean expression and is used to denote a transition from one node to another. The transition occurs immediately if the boolean condition evaluates to true.

```
state1 <- Map1.isEmpty();
```

3.3.5 CONCURRENCY

A concurrency statement consists of a map id or declaration, a colon, followed by a list of function call statements

separated by commas. This type of statement is used to call several sub-DFAs at once, each of which will step concurrently as they run, i.e. one transition at a time. This guarantees that a DFA called within this list will not follow a transition to a new node until every other DFA in this list has followed as many transitions as it has.

The map given at the beginning of this statement must be of type <String,String> and, at all points in the running of the DFAs, will contain a mapping of "DFAname -> current node of DFA". This map is then sent as an implicit argument to all DFAs in the list so that it may be polled but not edited. The statement below, for example, would ensure that DFA1, DFA2 and DFA3 would all step concurrently, and they would all have access to the entries in m.

```
Map <String, String> m : DFA1(), DFA2(arg1), String s
= DFA3(arg2, arg3);
```

3.3.6 RETURN

A return statement consists of the return keyword followed by an expression.

```
return i < 4;
```

3.4 SCOPE

Scope in StateMap is divided into local and global types. Local scope is particular to a node where global scope is particular to a DFA.

A variable declared within the curly braces of a DFA is accessible anywhere within that DFA, but not in functions (sub-DFAs called by that DFA. Arguments must be used to pass variables between DFAs.

A variable declared within the curly braces of a node is only accessible within that node.

4. TYPE

4.1 TYPE DECLARATION

In StateMap, it is required to explicitly declare type when declaring a variable or DFA (except main). The type of a variable will not change during the lifetime of that variable, i.e. StateMap is statically typed. The type of a DFA denotes the type that is returned when that DFA is called.

4.2 FUNDAMENTAL TYPES

4.2.1 INT

A 32-bit integer.

4.2.2 BOOLEAN

A boolean type that is either true or false.

4.2.3 DOUBLE

A 64-bit floating point number.

4.2.4 FLOAT

A 64-bit signed floating point number including an exponent portion.

4.2.4 CHAR

An 8-bit character.

4.2.5 STRING

A sequence of characters.

4.2.6 STACK

Normally considered a "non-fundamental" data type, but they are fundamental in StateMap because of their connection to DFAs. Must be declared with a type as follows:

```
stack<int> s;
```

Stacks, on the fundamental level, support the following operations:

peek - return the item on the top of the stack

pop - remove and return the item on the top of the stack

push - push a given item in the top of the stack

4.2.7 VOID

While not a type used in variable declaration, DFAs can have return type void if they do not return anything.

4.3 NON-FUNDAMENTAL TYPES

4.3.1 MAP

StateMap's version of a dictionary. Must be declared with two types as follows:

```
map<string, int> m;
```

The map functions are included in the standard library and allow users to add entries, delete entries, return values for given key, return keysets, etc.

5. STANDARD LIBRARY

The standard library provides an unordered map implementation.

5.1 DECLARING A MAP

Map<key, value> foo

5.2 INSERTING INTO A MAP

insert(key, value)

This function returns true if successful. Otherwise, it returns false.

5.3 DELETING FROM A MAP.

erase(key)

This function returns true if successful. Otherwise, it returns false.

5.4 FINDING A KEY

find(key)

This function returns the key's value if the key is valid. Otherwise, it returns void.

5.5 FINDING THE SIZE OF THE MAP

size()

5.6 DELETING THE ENTIRE MAP

clear()

This function returns true if successful. Otherwise, it returns false.

7. PROGRAM EXECUTION

Programs are run via command line, in the format:

```
./{PATH TO DIRECTORY}/{NAME OF EXECUTABLE} {ARGUMENTS}
```

For example:

```
./sorts/quicksort 0 9 2 3
```