A Haskell Library for Automata Theory

Ari Zerner
advised by Prof. Matthew Fluet
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY 14586
afz8559@cs.rit.edu

Abstract—This paper describes HAutomata, a Haskell library for automata theory.

Index Terms—Haskell; Automata theory

I. INTRODUCTION

This paper describes HAutomata, a Haskell library for automata theory. The code for HAutomata can be found at https://github.com/Ari-Zerner/hautomata. HAutomata is intended to be a comprehensive implementation of automata theoretic constructs, with an emphasis on pedagogical applications. Although development remains a work in progress, the goal of this project is for HAutomata to eventually supersede the functionality of all other automata theory libraries written in Haskell.

II. BACKGROUND

A. Automata Theory

Automata theory is a branch of theoretical computer science that studies abstract machines known as automata. An automaton recognizes a formal language by accepting only the strings in that language. There are several types of automata, each with different mechanisms and capability to recognize languages. HAutomata currently implements three commonly studied types: finite state automata, pushdown automata, and Turing machines.

B. Haskell

Haskell is a purely functional programming language originally released in 1990. It has a powerful static type system with features such as algebraic data types, type classes, and parametric polymorphism. Haskell uses lazy evaluation by default.

C. Existing Libraries

Haskell libraries for automata theory do exist. Examples include HaLeX [1] and turingMachine [2]. HaLeX provides an in-depth implementation of finite state automata and regular expressions, while turingMachine is less in-depth, but covers finite state automata, pushdown automata, and Turing machines.

III. DESIGN

A. Design Goals

The following design goals are applied throughout the design of HAutomata.

1) Consistency: Similar functionality should have similar interfaces. In order to achieve this goal, type classes are used where applicable, in order to enforce interface consistency using Haskell’s type system. Where it does not make sense to use type classes, similarity of interfaces is maintained manually.

2) Polymorphism: The library should be agnostic about types where possible, leaving users free to choose the types of their data. In order to achieve this goal, every automaton in HAutomata is parameterized with the types of its components. For example, finite state automata are parameterized with symbol and state types. This contrasts with turingMachine [2], which enforces the constraint that symbols are Unicode characters.

3) Implementation Independence: The public API should be unentangled with the underlying implementation, so that the public API can be designed for ease of use without constraining the implementation. In order to achieve this goal, data types are exported as blackboxes, so that users of the library cannot write code that depends on the implementation details of the types. Where appropriate, functions are written to allow controlled access to the underlying implementation, such that the underlying implementation can be changed without changing the public-facing signatures of the functions.

B. Modules

HAutomata is divided into four modules: Automata.Automaton, Automata.FSA, Automata.PDA, and Automata.TM.

1) Automata.Automaton: The main purpose of the Automaton module defines type classes, data types, and utility functions that are used by the different types of automata. Selected signatures from the module are shown in Listing 1. An instance of PartialDecider may be able to decide whether to accept or reject in its current state. PartialDecider is instantiated by all of the automata currently implemented in HAutomata. Decider is similar, but a Decider must be able to make the decision. Turing machines do not instantiate Decider.
Accepter and Steppable, like PartialDecider, are instantiated by every automaton in HAutomata. An instance of Accepter can decide whether to accept a list of inputs, and an instance of Steppable may be able to take a step when given an input. runSteppable is a utility function which, starting with a Steppable instance, steps through a list of inputs.

Listing 1. Selected signatures from Automata.Automaton

```haskell
data Decision = Reject | Accept

data PartialDecision = Undecided | Decided Decision

class PartialDecider a where
  partialDecide :: a -> PartialDecision

class PartialDecider a where
  decide :: a -> Decision

class Accepter input a | a -> input where
  accepts :: a -> [input] -> Bool

class Steppable input a | a -> input where
  step :: input -> a -> Maybe a

runSteppable :: Steppable input a
   => a -> [input]
   -> (Maybe input, a)
```

2) Automata.FSA, Automata.PDA, and Automata.TM: Each of these models implements one type of automaton: finite state automata, pushdown automata, and Turing machines, respectively. All follow a similar pattern, pursuant to the design goal of consistency. For each type of automaton, a deterministic version and a non-deterministic version is implemented, as well as a wrapper structure which may contain either. Constructors, accessors, and type class instances are provided for both the deterministic and non-deterministic version. For the wrapper structure, functions are provided for wrapping and unwrapping the specific versions, and the type class instances of the specific versions are lifted to the wrapper structure.

C. Design Trade-offs

1) NPDA Epsilon Transitions: Epsilon transitions are prohibited for non-deterministic pushdown automata. Whereas a non-deterministic finite state automata with epsilon transitions can take a step by calculating the epsilon closure of its states after it reads a symbol, an epsilon transition on a non-deterministic pushdown automaton can push to the stack, and the epsilon closure can thus be infinite. Future work may include a function to convert a non-deterministic pushdown automaton with epsilon transitions to one without.

2) Turing Machine Steps: The step operation for Turing machines is somewhat awkward. In order to instantiate the Steppable type class, the step operation must take an input. However, unlike finite state automata and pushdown automata, which step through their inputs one symbol at a time, a Turing machine loads its whole input onto its tape, and steps require no further input. Therefore, the step function on Turing machines must be given a unit (empty tuple) as input.

3) Decider and PartialDecider: It would be more elegant to have a single, unified type class for decision-making, rather than Decider and PartialDecider. However, taking either of the two as the canonical class would have significant drawbacks. If Decider were chosen, Turing machines would not be able to instantiate it, as a Turing machine is not able to make a decision at every step. On the other hand, if PartialDecider were chosen, the totality of decision making by finite state automata and pushdown automata could not be enforced by the type system.

IV. Verification

Verification is performed using The Haskell Test Framework [3] with QuickCheck [4]. Tests are divided into three modules, corresponding to the three source modules which implement automata: Test.Automata.FSA, Test.Automata.PDA, and Test.Automata.TM. HAutomata contains roughly as many lines of test code as lines of source code.

V. Future Work

HAutomata is a work in progress, and much more functionality is needed before it achieves the goal of becoming the foremost Haskell library for automata theory. Some possible extensions are listed here.

- More convenience functions and custom operators, in order to simplify the syntax of using HAutomata
- Functions to convert between the different types of automata
- Operations such as equivalence, union, intersection, and concatenation (for appropriate types of automata)
- Validation on automata constructors
- Where appropriate, functions to generate, rather than accept, the languages defined by automata
- Other types of automata, such as cellular automata and tree automata
- Automata wrappers such as a step counter
- Optimization of operations, especially step

REFERENCES