Pecan

An Automated Theorem Prover

Reed Oei, Eric Ma, Tatum Schmidt, Abdullah Dean,
Christian Schulz, Philipp Hieronymi

January 23, 2020

1 Introduction

Pecan is a system for *automated theorem proving* that represents logical predicates using Büchi automata. An **automated theorem prover** is a program that takes as input a statement and **decides** (i.e., proves or disproves) it. Theorem provers can be very useful: computers are reliable, and they never get tired or bored, allowing us to quickly explore new ideas. Though it is impossible to decide *all* statements, we can still use theorem provers to solve many interesting problems.

Pecan was largely inspired by Walnut [3], another automated theorem prover which uses finite automata rather than Büchi automata.

Praline is a scripting language designed to make working with Pecan more pleasant; it is so named because it is essentially syntactic sugar for Pecan.

The aim of the manual is to give sufficient information to introduce Pecan and Praline and to provide a reference for them.

Section 2 describes the mathematical background necessary to understand how Pecan works. This section is not strictly necessary to use Pecan, so you may wish to skip directly to Usage (Section 3) which contains several examples, for a quicker start. Section 3 gives a short tutorial and several examples of Pecan programs. Section 4 provides a reference how all of the various features of Pecan work. Section 5 describes at a high level how Pecan is implemented, the compilation process, and the various optimizations the system performs. Section 6 describes the Pecan Automata Format. Section 7 describes the Praline scripting language.

2 Background

Below is an informal introduction to automata and notation that will be used throughout this manual. For a formal introduction, see [2].

A word is a (possibly infinite) sequence of symbols from a alphabet (usually finite) Σ. A set of words is called a **language**. The set of all finite words over a given alphabet Σ is denoted by Σ^*, and the set of all infinite words over the same alphabet is Σ^ω. A **finite automaton** is a "machine" that accepts some **finite word**, or string of letters, such as "abc." We say an automaton accepts a language $L(\mathcal{A})$ if and only if every string in the language is accepted by the automaton and every string not in the language is rejected by it. They can be represented as a *finite* collection of **states** and **transitions**.

Automata have nice closure properties: we can combine them using first-order logic operations. Therefore, there is an algorithm that can decide any statement expressed solely in terms of first order logic operations and properties defined by finite automata.

Theorem 2.1. *[2] Let $\mathcal{A}$ and $\mathcal{B}$ be two finite automata. Then we can construct automata $\mathcal{A} \wedge \mathcal{B}$, $\mathcal{A} \vee \mathcal{B}$, and $\neg\mathcal{A}$, where $\mathcal{A} \wedge \mathcal{B}$ is the automata that accepts strings only if they are accepted by both $\mathcal{A}$ and $\mathcal{B}$, and so on.*

For the quantifiers, $\forall$ and $\exists$, we need to define automata that accepts multiple inputs.

Definition 2.2. An automaton $\mathcal{A}$ accepts a pair of words $(x_1 x_2 \dots x_n, y_1 y_2 \dots y_n)$ if it accepts the word $(x_1, y_1)(x_2, y_2)\dots(x_n, y_n)$.

We generalize this definition to n-tuples in the natural way.

Theorem 2.3. *[2] If $\varphi(x, y)$ is a predicate with two inputs, and automaton $\mathcal{A}$ over an alphabet Σ accepts pairs of words (x, y) that makes $\varphi(x, y)$ true then we can construct an automaton which accepts words that makes $p\varphi'(y) = \exists x \in \Sigma.\varphi(x, y)$ true.*

Note that we can express constructs such as $P \implies Q$ as $\neg P \vee Q$ and $\forall x.P(x)$ as $\neg(\exists x.\neg P(x))$. Therefore, we can combine predicates using first-order quantifiers and construct the automaton associated with the predicate.

Büchi Automata are an extension of the concept of finite automata to infinite inputs; they accept words that visit an accepting state infinitely often [2]. When we say "automata" with qualification through the rest of the manual, we are referring to Büchi automata. Importantly, the languages that Büchi automata define are closed under intersection, union, projection, and complementation, and emptiness checking is decidable [2]. Using Büchi automata has a number of advantages over finite automata:

- Some problems are more naturally expressed with Büchi automata.
- Some problems can *only* be expressed with Büchi automata.
- We can still express properties about finite strings (e.g., by using some symbol as an "end of input").

A major application of Büchi automata is in model checking: verifying that programs have the expected behavior.

3 Usage

To install Pecan see the instructions at `https://github.com/ReedOei/Pecan`.

3.1 Running Pecan

The standard method of using Pecan is to create a new file (with a `.pn` extension) which will hold all definitions and theorems. Then the file can be run using the following command on most operating systems (whether you use `python3` or `python` depends on how Python is installed):

```
$ python3 pecan.py FILENAME
```

You can also run the file in **interactive mode**, which will run the file and then allow you to directly type new definitions and directives with the definitions from the file loaded:

```
$ python3 pecan.py -i [FILENAME]
```

3.2 A First Theorem

For a taste of Pecan's syntax (see Section **??** for a complete description) and the sort of theorems we can prove using Pecan, we'll prove a simple theorem about natural numbers:

Theorem 3.1. *Every natural number is either even or odd.*

Pecan has a built-in definition of natural numbers, so we don't need to write that, but we will write a definition of "even" for natural numbers [1]. The primary method of defining predicates in Pecan is the following:

```
PREDICATE_NAME(ARGUMENTS) := BODY
```

In our particular case, we can define a number to be even if there is another number that is half of it. We write this definition in Pecan as:

```
is_even(x is nat) := exists y is nat. x = 2*y
```

Let's unpack this definition a little. First, we say that `x is nat` in the arguments of `is_even` so that Pecan knows the **type** of the arguments, and we do the same on the existential quantifier for `y`. Specifying the type allows Pecan to lookup the appropriate arithmetic operator definitions (e.g., for addition). You can think of `x is nat` as being equivalent to $x \in \mathbb{N}$. The body of the predicate, `exists y is nat.x =2*y` works almost exactly like in a standard mathematical definition; `is_even(x)` is true if and only if `exists y is nat.x =2*y`.

Note that instead of writing `x is nat` all the time, we can **restrict** certain variables to only be a specific type using the `Restrict` directive. Let's do that for a few variables:

```
Restrict x,y,z are nat.
```

Note that above `are` is just another way to write `is`. We can get an example of a number that this predicate accepts with the following line.

```
Display example natFormat { is_even(x) }.

[(x,0)]
```

That is, one input that `is_even` accepts is 0, which makes sense, because 0 is a natural number and 0 is even. Natural numbers in Pecan are, by default, in LSD (least-significant digit first) format, which is the reverse of the standard order; this order makes the underlying implementation significantly easier. We can ask Pecan for more interesting examples as well. For example, we can ask for an odd number greater than 33 by writing the following:

```
is_odd(x) := exists y. x = 2*y+1
Display example natFormat { x > 33 & is_odd(x) }.
```

We omitted the `x is nat` and `y is nat`; this omission is allowed because of the restriction we placed on `x` and `y`. When we run the file, Pecan responds with:

```
[(x,49)]
```

Again, this answer makes sense: it is an odd number, and it is greater than 33. This particular feature is non-deterministic, so you may not get the same examples inputs that I do. This output has been pretty-printed—if you want the raw input string, which will be an infinite binary string, you can instead use:

```
Display example stdFormat { x > 33 & is_odd(x) }.
```

[1] Pecan also has an even predicate in its standard library, called even, however, we will write our own for instruction purposes.

This command gives me `[(x,100011(0)^w)]`, meaning that the input string starts with 100011 and then is 0 repeated infinitely many times.

Now that we've explored Pecan a little bit, let's write the actual theorem, and ask Pecan to check it (specifically, we assert that it is true). Note that "theorems" are just predicates with no arguments—they will either be always true, or always false.

```
all_even_or_odd() := forall x. is_even(x) | is_odd(x)
#assert_prop(true, all_even_or_odd)
```

Pecan proves the theorem is true (had it failed, the output would be red):

```
[INFO] Checking if all_even_or_odd is true.
all_even_or_odd is true.
```

Of course, we can see this theorem is true in another way, by confirming that a natural number is odd if and only if it is not even.

```
odd_iff_not_even() := forall x. is_odd(x) <=> !is_even(x)
#assert_prop(true, odd_iff_not_even)
```

Again, Pecan confirms this theorem:

```
[INFO] Checking if odd_iff_not_even is true.
odd_iff_not_even is true.
```

Below are some more involved examples with somewhat less in-depth explanation. Note that Pecan allows the use of Unicode characters such as $\in$, $\exists$, and $\forall$, which we make use of frequently in the following sections for aesthetic reasons.

3.3 Examples

3.3.1 Example: The Chicken McNugget Problem

Henri Picciotto asked the following problem in his algebra textbook [4]:

> What is the greatest number of chicken nuggets that cannot be ordered using only boxes of 6, 9, and 20?

We call all such numbers **non-purchasable**, and we can define them in Pecan as follows:

```
Restrict n,m,a,b,c ∈ binary.
purchasable(n) := n ∈ binary ∧ ∃a. ∃b. ∃c. n = 6*a + 9*b + 20*c
non_purchasable(n) := n ∈ binary ∧ ¬purchasable(n)
```

We can then define the largest non-purchasable number in the natural way to be:

```
largest(n) := non_purchasable(n) ∧ ∀m. non_purchasable(m) ⟹ m ≤ n
```

Pecan produces the automaton in Figure 1 for us, which tells us that the largest non-purchasable number is 101011_2, or 43 in base 10.

Example: Properties of the Thue-Morse Word

Below, we develop a Pecan program that proves two well-known properties of the Thue-Morse word, T, which is defined by: the n-th digit of the Thue-Morse word, $T[n]$, is 1 if the binary representation of n has an odd number of 1's, and 0 otherwise [1].

The Thue-Morse word starts with: 11010011001011010010110011010 01…

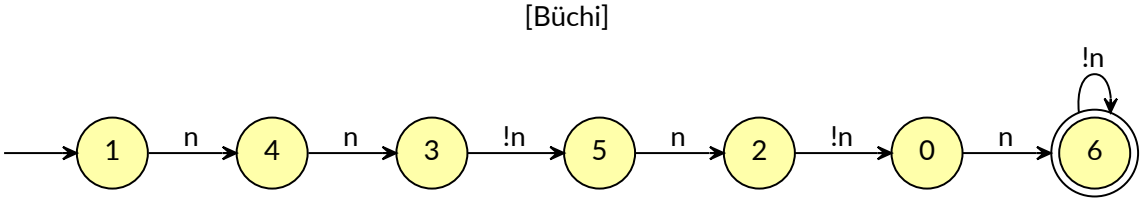

Figure 1: A Büchi automaton accepting 110101 in least significant digit first representation.

Definition 3.2. A word w is a **square** if it is of the form $w = xx$ for some nonempty word x. Similarly, w is a **cube** if it is of the form $w = xxx$ for some nonempty word x.

Theorem 3.3. *[1] The Thue-Morse word contains squares, but does not contain any cubes.*

Here is the equivalent definition of T in Pecan (odd_ones is defined as an automaton):

```
T(x) := odd_ones(x)
```

Below are definitions of square and cube for the Thue-Morse word in Pecan.

```
Restrict i,j,n ∈ binary.
square(i, n) := n > 0 ∧ T[i ... i+n] = T[i+n ... i+2*n]
cube(i, n) := square(i, n) ∧ square(i+n, n)
```

Finally, we state the theorems we would like to prove, and ask Pecan to attempt to prove that there are squares, but there are no cubes.

```
squares_exist() := ∃i. ∃n. square(i, n)
#assert_prop(true, squares_exist)
cubes_exist() := ∃i. ∃n. cube(i, n)
#assert_prop(false, cubes_exist)
```

The output from Pecan is the following, indicating that the theorem is true.

```
[INFO] Checking if squares_exist is true.
squares_exist is true.
[INFO] Checking if cubes_exist is false.
cubes_exist is false.
```

Here is another theorem about the Thue-Morse word that we can check.

Theorem 3.4. *[1] There are no overlapping squares, i.e. words of form $0x0x0$ or $1x1x1$ for some nonempty word x.*

We can express this theorem in Pecan as:

```
Restrict i,j,n ∈ binary.
o_square(i, n) := n > 0 ∧
                  T[i+1 ... i+n+1] = T[2+i+n ... 2+i+2*n] ∧
                  T[i] = T[i+n] ∧ T[i] = T[2+i+2*n]
o_squares_exist() := ∃i. ∃n. o_square(i, n)
#assert_prop(false, o_squares_exist)
```

Pecan verifies the theorem:

```
[INFO] Checking if o_squares_exist is false.
o_squares_exist is false.
```

Definition 3.5. A natural number p is a **period** of a word w if for all $0 \leq i < |w|$, $w_i = w_{i+p}$

Definition 3.6. A rational number e is an **exponent** of a finite word w if $e = |w|/p$ where p is a period of w.

Definition 3.7. The **critical exponent** of a word w is the supremum of all exponents of subwords of w.

Theorem 3.8. *The critical exponent of the Thue-Morse word is* 2.

Proof. As proven by Pecan, there are squares in the Thue-Morse word, so it is possible to achieve an exponent of 2. However, there are no overlapping squares, which means we cannot have an exponent greater than 2—a subword with an exponent greater than 2 will have more than 2 periods, which would be an overlapping square. □

This example demonstrates a common use case of theorem provers like Pecan: we can automate the tedious parts of a proof, allowing us to prove the theorem from a "high-level" perspective.

4 Features

4.1 Directives

```
#save_aut(FILENAME, PREDICATE)
#save_aut("bin_even.aut", bin_even)
```

Saves the predicate as an automaton in the HOA [5] format to the location specified.

```
#save_aut_img(FILENAME, PREDICATE_NAME)
#save_aut_img("bin_even.svg", bin_even)
```

Saves the predicate as an SVG file at the location specified.

```
#context(KEY, VALUE)
#context("adder", "bin_add")
```

Sets the key to the value in the current context. The context is a global key-value store that is used to manage some parts of Pecan such as defaults (e.g., the default adder, equals, etc.).

```
#end_context(KEY)
#end_context("adder")
```

Deletes the specified key from the context.

```
#load(FILENAME, FORMAT, PREDICATE)
#load("bin_add.aut", "hoa", bin_add(a,b,c))
#load("real/real_equal.txt", "pecan", real_equal(a, b))
```

Loads an automaton from a file with the specified name and arguments. Note that the number of arguments is *NOT* checked for you. In general, you should be careful when loading automata from files because there are not many safeguards (it is assumed you know what you are doing). There are three currently supported formats: "hoa" [5], as described earlier; "walnut", the format that the automated theorem prover Walnut [3] uses; and "pecan", a custom format that Pecan uses, described below in 6.

```
#assert_prop(PROP_VAL, PREDICATE_NAME)
#assert_prop(true, int_add_comm)
```

The main interface to the theorem proving capabilities of Pecan. The possible values for PROP_VAL are `true`, `false`, and `sometimes`. While theorems are typically represented by predicates that take no arguments, this directive still works if the predicate does take arguments. In that case, asserting a predicate is `true` is the same as asserting that it is true for all possible inputs; asserting that it is `sometimes` true checks if there is any input that it accepts.

```
#import(FILENAME)
#import("integers.pn")
```

Loads all definitions from the specified file into the current scope. Does *NOT* load restrictions into the current scope, but they function normally while evaluating the imported file. To control the search path, you can set the PECAN_PATH environment variable. By default, the current working directory (at the time of loading the program), the directory the current source file is in, and the library/ directory of the Pecan directory.

```
#forget(VARIABLE)
#forget(x)
```

Removes all restrictions on the specified variable.

```
#type(TYPE_PREDICATE, FUNCTION_PREDICATES)
#type(nat, {
    "adder": bin_add(any, any, any),
    "less": bin_less(any, any)
})
```

Defines a new type. TYPE_PREDICATE should be the part you write after the `is` in a restriction. For example, in `Restrict x is nat.`, the relevant part is `nat`; in `Restrict i is ostrowski(alpha).`, the relevant part is `ostrowski(alpha)`. The function predicates become available to be called using the names in quotes—this feature allows for ad-hoc polymorphism. It is also used to resolve arithmetic operators, such as `+` (which calls the relevant `adder`) and `<` (which calls the relevant `less`). For example, in the following code, `f` and `g` are equivalent.

```
Restrict a, b are nat.

f(a, b) := a < b
g(a, b) := bin_less(a, b)
```

The full list of special names is as follows:

`adder` Used to resolve `+`

`less` Used to resolve `<`

`zero` Used to resolve `0`

`one` Used to resolve `1`, and all integer constants other than `0` (e.g., `2` is calculated as `1 + 1`)

`equal` Used to resolve `=`

```
#shuffle(PREDICATE, PREDICATE, PREDICATE)
#shuffle(int_less(a, b), any2(a, b), integral_less(a, b))
```

Shuffles the two predicates into one, by alternating between the two; shuffling is defined as below.

Definition 4.1. The **shuffle** of two automata $\mathcal{A}$ and $\mathcal{B}$ is an automaton $\mathcal{C}$ such that if $a = a_1 a_2 \cdots$ and $b = b_1 b_2 \cdots$ are ω-words accepted by $\mathcal{A}$ and $\mathcal{B}$ respectively, then $c = a_1 b_1 a_2 b_2 \cdots$ is accepted by $\mathcal{C}$.

In the example above, `int_less(a, b)` accepts a and b such that $a < b$ where $a, b \in \mathbb{Z}$ and `any2(a, b)` accepts every pair of strings a and b. If we think of the shuffled string as a pair (e.g., if our string is $c = a_1b_1a_2b_2\cdots$, we could think of it as $c = (a_1a_2\cdots, b_1b_2\cdots)$) whose first component is the integral part of a real number and the second component is the fractional part, then their shuffle is an automaton that accepts real numbers a and b such that the integral part of a is less than that of b, and ignores the fractional part.

```
#shuffle_or(FILENAME, FORMAT, PREDICATE)
#shuffle_or(one_int(x), zeros(x), real_one(x))
```

This shuffle operation is the same as above, except that it only requires that one of the strings in the shuffle is accepted by one of the automata. More specifically:

Definition 4.2. The **(disjunctive) shuffle** of two automata $\mathcal{A}$ and $\mathcal{B}$ is an automaton $\mathcal{C}$ such that if $a = a_1a_2\cdots$ and $b = b_1b_2\cdots$ are ω-words such that *either* $\mathcal{A}$ accepts a *or* $\mathcal{B}$ accepts b (or both are accepted), then $c = a_1b_1a_2b_2\cdots$ is accepted by $\mathcal{C}$.

```
Restrict VARIABLES are TYPE.
```

In all following code in this file, the variables specified are now consider to be of the specified type, unless deleted by a `#forget`.

5 Implementation

[Describe the whole processing of compilation, type checking, etc.]

Pecan programs are compiled in a series of **transformations** from the syntax shown above into a simplified IR (Intermediate Representation). The process of executing a program is roughly as follows:

1. Parse the program into an AST (Abstract Syntax Tree).
2. Transform the program's AST into a simplified IR representation, rewriting constructs such as $\iff$ in terms of simpler ones, like $\wedge$, $\vee$, and $\neg$.
3. Load the standard library (if desired); loading the standard library requires going through all of the steps again, starting from step 1, but skipping this step.
4. Run the untyped optimizer, if enabled.
5. Perform type inference and run Praline code, if any, in the order that definitions appear in the program.
6. Perform a final IR lowering, eliminating several more constructs from the program IR, such as word ranges ($T[i..j] = T[k..\ell]$).
7. Run the typed optimizer, if enabled.
8. Evaluate the final program IR.

5.1 AST to IR Conversion

The following conversions are performed when converting the AST to the IR:

- $a \iff b$ becomes $(a \implies b) \wedge (b \implies a)$
- $a \implies b$ becomes $\neg a \vee b$

- $k \cdot x$ becomes $\overbrace{x + \cdots + x}^{k \text{ times}}$
- $a \neq b$ becomes $\neg(a = b)$
- $a > b$ becomes $b < a$
- $a \geq b$ becomes $b < a \vee a = b$
- $-a$ becomes $0 - a$
- $a \leq b$ becomes $a < b \vee a = b$
- $W[i] = W[j]$ becomes $W(i) \iff W(j)$
- $\forall x.P$ becomes $\neg(\exists x.\neg P)$

5.2 Optimization

5.2.1 Boolean Optimizations

The following boolean optimizations are performed:

- $\neg\neg P$ becomes P
- $\neg(P \wedge Q)$ becomes $\neg P \vee \neg Q$
- $\neg(P \vee Q)$ becomes $\neg P \wedge \neg Q$
- $\neg(x < y)$ becomes $y < x \vee x = y$
- $\top \wedge Q$ becomes Q
- $\bot \wedge Q$ becomes $\bot$
- $\top \vee Q$ becomes $\top$
- $\bot \vee Q$ becomes Q
- $x = x$ becomes $\top$

6 Pecan Automata Format

7 Praline

Praline is a scripting language intended to be a macro system that makes writing Pecan programs easier. It is a fairly "standard" functional language. The major features of Praline are:

- Built-in support for integers, strings, booleans, lists, and tuples.

```
Display 42^6. // 5489031744
Display "hello␣" ^ "world". // hello world
Display [1..5]. // [1,2,3,4,5]
Display snd ("test", 1). // 1
```

- Partial application

```
Define add x y := x + y.
Display map (add 10) [1..10]. // [11,12,13,14,15,16,17,18,19,20]
```

- Lambda functions:

```
Display (\i -> i^2) 10. // 100
```

- Pattern matching

```
Display match [1..10] with
        case fst :: snd :: _ -> fst + snd
        end. // 3
```

- Ability to interact easily with Pecan:

```
Display example natFormat { x > 33 & exists y. x=2*y }. // [(x,48)]
```

7.1 Usage

One of the main uses of Praline is to automate the process of checking certain sets of properties. For example, suppose we have a predicate `R(x, y)` which is supposed to represent some partial order on a type `T`, and some equality predicate `EQ(x, y)`. Ordinarily, we would have to write three predicates, and three assertions, as follows.

```
R_irrefl() := forall x is T. !R(x, x)
#assert_prop(true, R_irrefl)
R_antisym() := forall x is T. forall y is T. if R(x, y) & R(y, x) then EQ(x, y)
#assert_prop(true, R_antisym)
R_trans() := forall x is T. forall y is T. forall z is T.
               if R(x, y) & R(y, z) then R(x, z)
#assert_prop(true, R_trans)
```

However, using Praline, we can write a macro that generates these definitions for us, given `R`, `T`, and `EQ`. Such a macro exists in Pecan's standard library, which we can call as follows:

```
Execute partialOrderCheck R EQ T.
```

There are three commands for interacting with Praline:

```
Define function args := body.
```

Defines a new function with the specified arguments and body.

```
Display term.
```

Evaluates the term and then prints the result to the screen.

```
Execute term.
```

The same as `Display`, except the result is not printed to the screen.

7.1.1 Examples

Factorial Praline can also be used as a general purpose programming language, although that is not its main purpose. For example, the following is the factorial function defined in Praline:

```
Define factorial n :=
    if n = 0 then 1
    else n * factorial (n - 1).

Display factorial 24. // 620448401733239439360000
```

Integer formatting The `example` function is often very useful for debugging, but it can be confusing to see raw ω-words, so intead it is convenient to define **formats**, which can then be used as follows:

```
Display example myFormat { P(x) }.
```

The following program formats an accepting word (which has been converted to binary) as a natural number. Natural numbers in Pecan are of the form $x0^w$ where x is some binary string, because natural numbers are finite.

```
Define fromBinary := foldr (\a b -> a + 2*b) 0.

Define natFormat var prefix cycle :=
    if cycle = [0] then
        (var, if isEmpty prefix then "0" else toString (fromBinary prefix))
    else
        stdFormat var prefix cycle ^ "␣(NOT␣A␣VALID␣NATURAL␣NUMBER)"
```

Note that `stdFormat` formats the accepting word as an ω-word.

Graphing The following program can be used to obtain a subset of the graph of some function defined in Pecan.

```
Define graphPoint format F x :=
    let y be { F(x, y) } in
    (x, lookup "y" (example format y)).
Define graph format F := map (graphPoint format F).

Restrict x, y are nat.
double_f(x, y) := 2*x = y
Display graph natFormat double_f [1,2,3,4]. // [(1,2),(2,4),(3,6),(4,8)]
```

7.2 Features

7.3 Builtins

`check pecanTerm`

Returns `true` if the input term is always true, and `false` otherwise.

`toString term`

Converts the evaluated term to a string.

`print term`

Prints the term to the screen, and returns `true`.

`emit pecanTerm`

Outputs the specified Pecan term as a definition in the current location in the file and returns `true`.

`freshVar`

Returns a string representing a fresh (i.e., unused variable name).

`acceptingWord pecanTerm`

Returns an associative list mapping variable names to **accepting words**, a pair of a **prefix** and a **cycle**. For example, the accepting word `([true,false,true], [false])` represents the ω-word 1010^ω.

`toChars str`

Converts `str` into a list of strings representing the characters in the string. For example, `toChars "blah"=["b", "l", "a", "h"]`.

`cons head tail`

Constructs a list starting with `head` and ending with `tail`.

References

[1] Jean-Paul Allouche and Jeffrey Shallit. *Automatic Sequences: Theory, Applications, Generalizations*. Cambridge University Press, 2003. DOI: 10.1017/CBO9780511546563.

[2] Bakhadyr Khoussainov and Anil Nerode. *Automata Theory and Its Applications*. Secaucus, NJ, USA: Birkhauser Boston, Inc., 2001. ISBN: 3764342072.

[3] Hamoon Mousavi. "Automatic Theorem Proving in Walnut". In: *CoRR* abs/1603.06017 (2016). arXiv: 1603.06017. URL: http://arxiv.org/abs/1603.06017.

[4] H. Picciotto and A. Wah. *Algebra: Themes, Tools, Concepts – Teachers' Edition*. Creative Publications, 1994. ISBN: 9781561072521. URL: https://books.google.com/books?id=%5C_cOhD13J3ZMC.

[5] *The Hanoi Omega-Automaton Format*. http://adl.github.io/hoaf/. Accessed: 2020-01-22.