Pecan

An Automated Theorem Prover

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

April 6, 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 [8], 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 5 provides a reference how all of the various features of Pecan work. Section 4 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 [6].

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 a finite 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**.

Finite automata have nice closure properties: we can combine them using first-order logic operations, because they are closed under intersection, union, complementation, and projection. 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. *[6] Let $\mathcal{A}$ and $\mathcal{B}$ be two finite automata. Then we can construct the automata accepting the languages $L(\mathcal{A}) \cap L(\mathcal{B})$, $L(\mathcal{A}) \cup L(\mathcal{B})$, and $L(\mathcal{A})^c$.*

For the quantifiers, $\forall$ and $\exists$, we define automata accepting 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 can generalize this definition to n-tuples in the natural way.

Theorem 2.3. *[6] 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 $\varphi'(y) = \exists x \in \Sigma.\varphi(x, y)$ 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 [6]. 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 [6]. 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 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) := ∃ 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, `∃ y is nat.x =2*y` works almost exactly like in a standard mathematical definition; `is_even(x)` is true if and only if `∃ 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) := ∃ 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() := ∀ 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() := ∀ 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 [10]:

> 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…

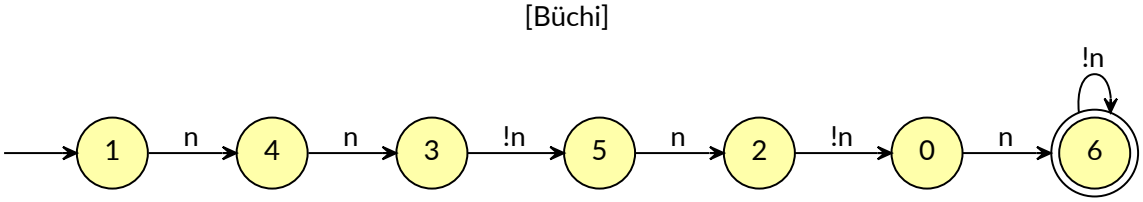

Figure 1: A Büchi automaton accepting 110101 in least significant digit first representation.

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 $0 \le i < |w|$, $w_i = w_{i+p}$

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 Implementation

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. Interpret the final program IR.

4.1 IR Syntax

Prog ::=

4.2 Type Checking

A *type* in Pecan is represented by a Büchi automaton. We say that $x : \tau$, pronounced "x has/is of type τ" when $x \in L(\tau)$, sometimes written $\tau(x)$ holds. Additionally, types may be *partially applied*, written $\tau = P(x_1, \ldots, x_n)$, where P is some Büchi automaton. Then $y : \tau$ when $(y, x_1, \ldots, x_n) \in L(P)$; and $\tau(y)$ holds when $y : \tau$. In the concrete syntax of Pecan, we write `y is tau` or `y ∈ tau`; for one or more variables, we can write `x, y, z are tau` to mean $x : \tau$, $y : \tau$, $z : \tau$. Finally, there is a special type called `Inferred`, which should be thought of as a type that will be discovered later by

how the value is used; or, as its name suggests, the type will eventually be **inferred** from context. $L(\texttt{Inferred})$ is undefined—`Inferred` is NOT represented by an automata.

The judgement $\Gamma \vdash x : \tau$ means that we can prove $\tau(x)$ is true in the environment Γ, which is a set of assumptions $x : \tau$. The judgement $\Gamma \vdash P$ prop means that P is a well-formed proposition in the environment Γ. A predicate $P(\overline{x : \tau}) := Q$ is well-formed when $\overline{x : \tau} \vdash Q$ prop.

We typecheck a sequence of Pecan predicates in order, and assume below that the set of all well-formed predicates is ambiently available as $\mathcal{P}$.

Structures In order to provide a mechanism for ad-hoc polymorphism, Pecan allows the definition and use of *structures*. This feature also facilitates the use of nicer syntax for arithmetic expressions (e.g., $x + (y + z) = w$ instead of $\exists t.\texttt{adder}(x, y, t) \wedge \texttt{adder}(t, z, w)$) without tying ourselves to a single numeration system. For example, `adder` will be resolved to some concrete predicate predicate based on the type of x, y, and z. The exact rules and definitions related to this feature are given below. We assume that structure definitions are ambiently available throughout the program.

Definition 4.1. A *structure* is a pair $(t(x_1, \ldots, x_n), D)$ where each x_i is an identifier, such that for some $\tau_1, \ldots, \tau_n$, $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash t(x_1, \ldots, x_n)$ prop and D is a map of identifiers to *call templates*; that is, it is of the form $f(y_1, \ldots, y_m)$, where each y_i may be either: 1) x_j for some j or 2) $*$, which is pronounced "any."

The *name* of the structure is t.

We write the sequence of indexes of the arguments that are $*$, called *parameters*, as $\texttt{params}(f(y_1, \ldots, y_m))$. A call template is called n-ary if $|\texttt{params}(f(y_1, \ldots, y_m))| = n$. The sequence of the indexes of the other arguments, which are not $*$, called *implicits*, is written $\texttt{implicit}(f(y_1, \ldots, y_m))$. For example, $\texttt{params}(f(a, *, *, b, *, *)) = [2, 3, 5, 6]$ and $\texttt{implicit}(f(a, *, *, b, *, *)) = [1, 4]$. We write $\texttt{params}(f(y_1, \ldots, y_m))[i]$ (resp. $\texttt{implicit}(f(y_1, \ldots, y_m))[i]$) to denote the i-th parameter (resp. implicit).

We assume that typechecking has been done before evaluating, because we may need structure information at run-time to resolve *dynamic calls*, that is, predicate calls whose name matches some definition inside a type. We denote the type that an expression e got when typechecking by $\text{typ}(e)$.

We write $t[P] = Q(y_1, \ldots, y_m)$ to look up a definition in the associated map D, and we say that t *has a definition* for P in this case. If t does not have a definition for P, then we write $t[P] = \bot$.

Definition 4.2. A structure is called *numeric* if it has a ternary definition for `adder` and a binary definition `less`. We write $x + y = z$ when $\texttt{adder}(x, y, z)$ holds and $x < y$ when $\texttt{less}(x, y)$ holds.

A numeric structure may also optionally contain the following definitions; otherwise a default predicate applies.

> A binary definition `equal`, written $x \equiv y$. If not provided, the default is simply standard equality, $x = y$.
>
> A unary definition `zero`. If not provided, the default is 0^ω.
>
> A unary definition one. If not provided, the default is x such that $0 \leq x \wedge \forall y.y = 0 \vee x \leq y$.

Definition 4.3. The *predecessor* of two types τ and σ, written $\tau \wedge \sigma$ is given by following partial function:

$$\tau \wedge \sigma = \begin{cases} \sigma & \text{if } \tau = \texttt{Inferred} \text{ and } \sigma \text{ is numeric} \\ \sigma & \text{if } \forall \mathcal{V}(\tau).\forall x.\tau(x) \implies \sigma(x) \\ \tau & \text{if } \forall \mathcal{V}(\tau).\forall x.\sigma(x) \implies \tau(x) \\ \tau & \text{if } \sigma = \texttt{Inferred} \text{ and } \tau \text{ is numeric} \end{cases}$$

Definition 4.4. We can *resolve* a call $P(e_1, \ldots, e_n)$ as $Q(a_1 : \tau_1, \ldots, a_m : \tau_m)$, written $P(e_1, \ldots, e_n) \rightsquigarrow Q(a_1 : \tau_1, \ldots, a_m : \tau_m)$, if $Q(a_1 : \tau_1, \ldots, a_m : \tau_m) \in \mathcal{P}$ and for some structure $t(x_1, \ldots, x_\ell)$, for each $1 \leq i \leq n$, either:

1. $\text{typ}(e_i) = t(x_1, \ldots, x_\ell)$, and $t[P] = Q(b_1, \ldots, b_m)$ such that for each $1 \le j \le m$,

$$a_j = \begin{cases} x_k & \text{if } \texttt{implicit}(Q(b_1, \ldots, b_m))[k] = j \\ e_k & \text{if } \texttt{params}(Q(b_1, \ldots, b_m))[k] = j \end{cases}$$

2. $\text{typ}(e_i) = s(y_1, \ldots, y_p)$, where $s \neq t$ and $s[P] = \bot$.

$\boxed{\Gamma \vdash e : \tau}$ **Typing**

$$\frac{i \in \mathbb{Z}}{\Gamma \vdash i : \texttt{Inferred}}\ \textsc{Int} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{Var} \qquad \frac{\Gamma \vdash a : \tau \qquad \Gamma \vdash b : \sigma}{\Gamma \vdash a \oplus b : \tau \wedge \sigma}\ \textsc{Op} \quad \text{where } \oplus \in \{+, -, /\}$$

$$\frac{\forall i \in \{1, \ldots, n\}. \Gamma \vdash e_i : \tau_i \qquad f(x_1 : \tau_1, \ldots, x_n : \tau_n, r : \tau) \in \mathcal{P}}{\Gamma \vdash f(e_1, \ldots, e_n) : \tau}\ \textsc{Func}$$

$\boxed{\Gamma \vdash P \text{ prop}}$ **Well-formed Propositions**

$$\frac{\Gamma \vdash a : \tau \qquad \Gamma \vdash b : \sigma \qquad \tau \wedge \sigma \neq \texttt{Inferred}}{\Gamma \vdash a \bowtie b \text{ prop}}\ \textsc{Rel} \quad \text{where } \bowtie \in \{=, <\}$$

$$\frac{\Gamma \vdash P \text{ prop} \qquad \Gamma \vdash Q \text{ prop}}{\Gamma \vdash P \oplus Q \text{ prop}}\ \textsc{BinPred} \quad \text{where } \oplus \in \{\vee, \wedge\} \qquad \frac{\Gamma \vdash P \text{ prop}}{\Gamma \vdash \neg P \text{ prop}}\ \textsc{Comp}$$

$$\frac{\Gamma, x : \tau \vdash P \text{ prop}}{\Gamma \vdash \exists x : \tau. P \text{ prop}}\ \textsc{Exists} \qquad \frac{\forall i \in \{1, \ldots, n\}. \Gamma \vdash e_i : \tau_i \qquad f(x_1 : \tau_1, \ldots, x_n : \tau_n) \in \mathcal{P}}{\Gamma \vdash f(e_1, \ldots, e_n) \text{ prop}}\ \textsc{Call}$$

4.3 Evaluation

The Pecan interpreter is a simple tree-walking interpreter which typechecks, then processes, each top-level construct (see Section 4.1) in sequential order.

Operations with non-trivial implementations are described in detail below. Most basic automata operations (e.g., conjunction, disjunction, complementation, emptiness checking, simplification) are implemented using the Spot library [4], so details of these algorithms are not presented here.

4.3.1 Automata Representation

Automata are represented by a pair of $(V, \mathcal{A})$, where V is a map taking variable names to an ordered list of APs that represent it, called the *variable map*, and $\mathcal{A}$ is a Spot automaton (specifically, a value of type `spot.twa_graph`). For more information about this underlying representation, see the Spot library [4].

Variable Maps A *variable map* is a essentially a set of mappings $x \mapsto [\text{ap}_1, \ldots, \text{ap}_n]$.

We denote by $V[x]$ the list of APs that x is represented by, and we denote by $V \cup W$ their union, which is only defined when the only keys that V and W have in common have identical APs. $V \sqcup W$ is the disjoint union of these maps.

For two variable maps V and W, $V \ll W$ denotes their *biased merge*, which is a pair (U, θ) of a variable map U and a substitution θ such that $U = V \cup W\theta$. A substitution is a set of mappings $a \mapsto b$ where a and b are both APs, which can be applied to a variable map or an automaton to rename the APs in them. For example, if $\theta = \{a \mapsto d, c \mapsto e\}$, then $\{x \mapsto [a, b, c]\}\theta = \{x \mapsto [d, b, e]\}$.

4.3.2 Logical Operations

Fundamental automata operations (i.e., $\wedge$ and $\vee$) are defined so:

$$(V,\mathcal{A}) \oplus (W,\mathcal{B}) = \begin{cases} (V \ll W, \mathcal{A} \oplus \mathcal{B}) & \text{if } |S(\mathcal{A})| < |S(\mathcal{B})| \\ (W \ll V, \mathcal{A} \oplus \mathcal{B}) & \text{otherwise} \end{cases}$$

where $S(\mathcal{A})$ denotes the set of states of $\mathcal{A}$.

4.3.3 Predicate Calls

$$\frac{\begin{array}{cc} \forall i. e_i \Downarrow (\mathcal{A}_i, x_i) & \texttt{nonvar}(e_1,\ldots,e_n) = [k_1,\ldots,k_\ell] \\ P(x_1,\ldots,x_n) \rightsquigarrow Q(y_1,\ldots y_m) & Q(z_1,\ldots,z_m) := R \qquad R \Downarrow \mathcal{B} \end{array}}{P(e_1,\ldots,e_n) \Downarrow \exists x_{k_1},\ldots,x_{k_\ell}.\mathcal{A}_1 \wedge \cdots \wedge \mathcal{A}_n \wedge \mathcal{B}[y_1/z_1,\ldots,y_m/z_m]} \textsc{Call}$$

where $\mathcal{B}[y_1/z_1,\ldots,y_m/z_m]$ denotes substituting each y_j for z_j in $\mathcal{B}$, defined in Section 4.3.7, and $\texttt{nonvar}(e_1,\ldots,e_n)$ denotes the nonvariable positions in $e_1,\ldots,e_n$.

The rule for evaluating predicate calls is rather complicated, due to the possibility of dynamic calls. Fundamentally, the rule does the following:

1. Evaluates each argument e_i as $(\mathcal{A}_\rangle x_i)$.
2. Resolves the call P as some predicate Q
3. Evaluates the body of Q, R and then substitutes in the variables y_i (in the resolve call, some of which will be the result variables x_i) as appropriate.
4. Projects out the intermediate result variables x_{k_p}, where the k_p are the nonvariable positions in $e_1,\ldots,e_n$.

4.3.4 Expressions

Denote by $\mathcal{V}(E)$ the list of free variables occurring in E. For example, $\mathcal{V}(a+b+c) = [a,b,c]$. Denote by $E[v/x]$ the expression E with v substituted for x where $x \in \mathcal{V}(E)$.

An expression E evaluates to a pair $(\mathcal{A},x)$ of an automaton $\mathcal{A}$ and a variable x where $\mathcal{A}$ accepts $(v_1,\ldots,v_n,x)$ such that $\mathcal{V}(E) = [x_1,\ldots,x_n]$ and $E[v_1/x_1,\ldots,v_n/x_n] = x$.

$$\frac{}{x \Downarrow (\top, x)} \textsc{Var} \qquad \frac{a+b=z \Downarrow \mathcal{A}}{a+b \Downarrow (\mathcal{A},z)} \textsc{Add} \text{ where } z \text{ is fresh} \qquad \frac{z+b=a \Downarrow \mathcal{A}}{a-b \Downarrow (\mathcal{A},z)} \textsc{Sub} \text{ where } z \text{ is fresh}$$

$$\frac{}{0 \Downarrow (\texttt{zero}(x),x)} \textsc{Zero} \text{ where } x \text{ is fresh} \qquad \frac{}{1 \Downarrow (\texttt{one}(x),x)} \textsc{One} \text{ where } x \text{ is fresh}$$

$$\frac{\overbrace{1+1+\cdots+1}^{n \text{ times}} \Downarrow (\mathcal{A},x)}{n \Downarrow (\mathcal{A},x)} \textsc{Int} \qquad \frac{f(e_1,\ldots,e_n,x) \Downarrow \mathcal{A}}{f(e_1,\ldots,e_n) \Downarrow (\mathcal{A},x)} \textsc{Func} \text{ where } x \text{ is fresh}$$

$$\frac{a \Downarrow (\mathcal{A},x) \qquad b \Downarrow (\mathcal{B},y)}{a=b \Downarrow \mathcal{A} \wedge \mathcal{B} \wedge (x \equiv y)} \textsc{Equal} \qquad \frac{a \Downarrow (\mathcal{A},x) \qquad b \Downarrow (\mathcal{B},y)}{a<b \Downarrow \mathcal{A} \wedge \mathcal{B} \wedge (x < y)} \textsc{Less}$$

4.3.5 Existential Quantification

To compute $\exists x \in \tau.P$, we co

For the actual implementation of the projection operation, see Section 4.3.6.

4.3.6 Projection

4.3.7 Substitution

4.4 AST to IR Conversion

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

- $a \iff b$ becomes $(a \implies b) \land (b \implies a)$
- $a \implies b$ becomes $\neg a \lor b$
- $k \cdot x$ becomes $\overbrace{x + \cdots + x}^{k \text{ times}}$
- $a \neq b$ becomes $\neg(a = b)$
- $a > b$ becomes $b < a$
- $a \geq b$ becomes $b < a \lor a = b$
- $-a$ becomes $0 - a$
- $a \leq b$ becomes $a < b \lor a = b$
- $W[i] = W[j]$ becomes $W(i) \iff W(j)$
- $W[i] \neq W[j]$ becomes $\neg(W[i] = W[j])$
- $W[i..j] = W[k..\ell]$ becomes $j + k = i + \ell \land \forall n \in \text{typ}(i).i + n < j \implies W[i + n] = W[k + n]$
- $W[i..j] \neq W[k..\ell]$ becomes $\neg(W[i..j] = W[k..\ell])$
- $\forall x_1 \in \tau_1, \ldots, x_n \in \tau_n.P$ becomes $\neg(\exists x_1 \in \tau_1, \ldots, x_n \in \tau_n.\neg P)$
- $\exists x_1 \in \tau_1, \ldots, x_n \in \tau_n.P$ becomes $\exists x_1 \in \tau_1 \ldots . \exists x_n \in \tau_n.P$

where $\text{typ}(i)$ denotes the type of i.

4.5 Optimization

4.5.1 Boolean Optimizations

The following boolean optimizations are performed:

- $\neg\neg P$ becomes P
- $\neg(P \land Q)$ becomes $\neg P \lor \neg Q$
- $\neg(P \lor Q)$ becomes $\neg P \land \neg Q$
- $\neg(x < y)$ becomes $y < x \lor x = y$
- $\top \land Q$ becomes Q
- $\bot \land Q$ becomes $\bot$
- $\top \lor Q$ becomes $\top$
- $\bot \lor Q$ becomes Q
- $x = x$ becomes $\top$

4.5.2 Arithmetic Optimizations

Note that these optimizations are **not** safe if you've defined arithmetic predicates that don't behave reasonably.

- $0 + x$, $x + 0$, and $x - 0$ become x
- $a + b = c$ becomes `adder`(a, b, c)

4.5.3 Unused Variables

- $\exists x.P$ becomes P if $x \notin fv(P)$

where $fv(P)$ denotes the free variables of P.

5 Features

5.1 Formulas

5.2 Expressions

Arithmetic

Functions Some predicates represent relations which are functions. For readability, Pecan allows the use of *placeholders*, which allow specifying the output of a function-like predicate. For example, if we have a predicate `bin_add(x, y, z)` accepting triples such that $x + y = z$ for binary numbers x, y, and z, then `bin_add(x, y, _)` is equivalent to writing $x+y$. So we could have written `bin_add(x, y, _) =z`. Such a formula will be translated into:

```
∃v. bin_add(x, y, v) & v = z
```

Because the last argument to a function-like relation is typically the output argument, we can simply write `f(x)` to mean `f(x, _)`.

5.3 Annotations

Pecan supports *annotations*, which may *wrapped* around any formula. They do **not** change the logical value of the automaton, but are instead used to tell Pecan how to evaluate a formula (e.g., when to simplify).

The follow annotations are available:

`@postprocess[FORMULA]`

Performs a series of simplifications on the automaton representing `FORMULA`.

`@simplify[FORMULA]`

Turns on simplification (the default) for the formula and all its subformulas.

`@no_simplify[FORMULA]`

Turns off simplification for the formula and all its subformulas.

`@simplify_states[FORMULA]`

Performs basic simplifications on states, such as merging equivalent states and purging unreachable and dead states.

```
@simplify_edges[FORMULA]
```

Performs basic simplifications on states with the goal of reducing the number of edges. See the Spot documentation for `merge_edges` for more information.

5.4 Automatic Words

Currently only binary automatic words are supported, which are essentially just syntactic sugar for predicates. Any predicate P can be interpreted as a word by writing $P[i]$, which is treated as 1 if $P(i)$ is true, and 0 if $P(i)$ is false. Then we have the following translations into the IR:

- $P[i] = 1$ becomes $P(i)$
- $P[i] = 0$ becomes $\neg P(i)$
- $P[i] = Q[j]$ becomes $P(i) \iff P(j)$
- $P[i] \neq Q[j]$ becomes $\neg(P(i) \iff P(j))$

5.5 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 [8] 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 5.1. 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 $c = a_1 b_1 a_2 b_2 \cdots$, we could think of it as $c = (a_1 a_2 \cdots, b_1 b_2 \cdots)$) 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 5.2. The **(disjunctive) 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 such that *either* $\mathcal{A}$ accepts a *or* $\mathcal{B}$ accepts b (or both are accepted), then $c = a_1 b_1 a_2 b_2 \cdots$ 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`.

6 Automata Formats

Below are details on all the automata formats that Pecan supports. Note that all formats support both deterministic and nondeterministic automata.

6.1 HOA

Pecan supports the HOA format [5], in which case when loading the parameter **must** be given the same names as the APs in the automaton, in the same order. This format only supports binary alphabets; to avoid this limitation you must either binary-encode your automaton yourself or use another format. Both the Walnut and Pecan formats support arbitrary integer base alphabets.

To use this format, write:

```
#load("filename.txt", "hoa", pred_name(arg1, arg2, ...))
```

An example is shown in Figure 2.

6.1.1 HOA (Modified)

Pecan also supports the HOA format as described above, except that the first line may be of the form:

```
VAR_MAP: PYTHON_DICT
// Example:
VAR_MAP: {'a': ['__ap76', '__ap77'], 'n': ['__ap78', '__ap79']}
```

where PYTHON_DICT is a valid Python dictionary mapping variable names to the ordered collection of APs (atomic propositions) that represents them. The rest of the file follows the HOA format.

```
HOA: v1
States: 2
Start: 1
AP: 1 "a"
acc-name: Buchi
Acceptance: 1 Inf(0)
properties: trans-labels explicit-labels state-acc complete
properties: deterministic stutter-invariant terminal
--BODY--
State: 0 {0}
[t] 0
State: 1
[!0] 0
[0] 1
--END--
```

Figure 2: An automaton in the HOA format, accepting words that contain at least one 0.

```
{0,1,2}
0 0
0 -> 0
0 -> 1
1 -> 0
2 -> 0
1 1
0 -> 1
2 -> 1
```

Figure 3: An automaton in the Walnut format that accepts words that eventually end in $(0^*2)^\omega$.

6.2 Walnut

Pecan supports the same format that Walnut [8] uses. Pecan uses the Unix standard of using files encoded using UTF-8 with no BOM (as this would be redundant for UTF-8), unlike Walnut which (at the time of writing) requires UTF-16BE and allows, but does not require, a BOM. This format supports automata in any integer base (i.e., those with an alphabet like $\{0,1,2,\ldots\}$).

To use this format, write:

```
#load("filename.txt", "walnut", pred_name(arg1, arg2, ...))
```

An example is shown below in Figure 3.

See the Walnut manual for details https://github.com/hamousavi/Walnut/blob/master/Automatic%20Theorem%20Proving%20in%20Walnut.pdf.

6.3 Pecan

Pecan also supports it's own automata format, which is based on the Walnut format but allows for more descriptive state names and comments. This format supports automata in any integer base (i.e., those with an alphabet like $\{0,1,2,\ldots\}$).

To use this format, write:

```
// Input is a natural number with each digit encoded in binary and separated by 2s.
{0,1,2}
// All inputs must start with 2
start: 0
2 -> zero,even
zero,even: 0
0 -> zero,even
2 -> zero,odd
zero,odd: 0
0 -> zero,odd
1 -> done,odd
2 -> zero,even
done,odd: 1
0 -> done,odd
1 -> done,odd
2 -> done,odd
```

Figure 4: An automaton in the Pecan format, accepting n such that the n-th digit of the characteristic Sturmian word (of any slope) is 1.

```
#load("filename.txt", "pecan", pred_name(arg1, arg2, ...))
```

Somewhat informally, the format is:

```
AUTOMATON ::= ALPHABET_LINE STATE*

ALPHABET_LINE ::= INPUT+
INPUT ::= "{" 0, 1, ... "}"

STATE ::= LABEL ":" (0 | 1) TRANSITION*
SYMBOL ::= INTEGER*
LABEL ::= [^:]+
TRANSITION ::= SYMBOL "->" LABEL

COMMENT ::= "//" .*
```

An example is shown in Figure 4.

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 & ∃ 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() := ∀ x is T. !R(x, x)
#assert_prop(true, R_irrefl)
R_antisym() := ∀ x is T. ∀ y is T. if R(x, y) & R(y, x) then EQ(x, y)
#assert_prop(true, R_antisym)
R_trans() := ∀ x is T. ∀ y is T. ∀ 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` directive is often very useful for debugging, but it can be confusing to see raw ω-words, so instead it is convenient to define **formats**, which can then be used as follows:

```
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

7.2.1 Directives

Praline has three primitive directives: `Alias`, `Execute`, and `Define`.

`Alias` allows defining **new** directives which are simply abbreviations for other directives, possibly even other aliases. See Section **??** for more details.

7.2.2 Aliases

The aliases defined in the standard library are:

```
Alias "Display" ==> Execute print .
Alias "Example" ==> Display uncurry example .
Alias "Theorem" ==> Execute uncurry theoremCheck .
```

To use these (as we have already in some of the above examples), we may write the following:

```
Display factorial 24.
Example (natFormat, { x > 10 }) .
```

These rewrite into:

```
Execute print (factorial 24).
Execute print (uncurry example (natFormat, {x > 10 })).
```

It is generally preferable to use `Theorem` as opposed to `#assert_prop`, because it allows for more descriptive names, like the following:

```
Theorem ("Addition of natural numbers is commutative.", {
    ∀ x, y are nat. x + y = y + x
}).
```

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"]`.

`split sep str`

`cons head tail`

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

`compare a b`

`equal a b`

```
mkAutomaton inputNames inputBases

addState aut stateLabel isAccepting

addTransition aut src dst syms

buildAut aut

writeFile path contents

readFile path

deleteFile path
```

8 Case Study: Sturmian Words

One of the interesting applications of a theorem prover like Pecan is its ability to decide properties of automatic words, such as Sturmian words. Not only can Pecan prove properties of such words, but it can also act as a (counter)example generator because of the constructive nature of its decision procedure: if property is **not** true, then Pecan can give a counterexample of exactly when it fails. In this section, we explore several properties of Sturmian words using Pecan.

8.1 Background

8.1.1 Finite and ω-Words

Let Σ^* denote the set of finite words on the alphabet Σ, let Σ^+ denote the set of nonempty finite words on the alphabet Σ, and let Σ^ω denote the set of ω-words on the alphabet Σ.

For a word w, let $w[i]$ denote the i-letter of w. Let $w(i,n)$ denote the length-n factor of w starting at i and ending at $i+n-1$, that is, $w[i\ldots i+n-1] = w[i]w[i+1]\cdots w[i+n-1]$. Let $|w|$ denote the *length* of w. Let w^R denote the *reverse* of w, so if $w = w[1]w[2]\cdots w[n]$, then $w^R = w[n]w[n-1]\cdots w[1]$. In a binary alphabet, $\Sigma = \{0,1\}$, for a symbol $x \in \Sigma$, let $\overline{x}$ denote the *complement* of x, so $\overline{0} = 1$ and $\overline{1} = 0$. For a word w over a binary alphabet, let $\overline{w}$ denote the *complement* of w, that is $\overline{w} = \overline{w[1]}\,\overline{w[2]}\cdots\overline{w[n]}$. A positive integer p is a *period* of a word w if $w[i] = w[i+p]$ for all i. A word is *periodic* if it has any period, and aperiodic if there are no periods.

8.1.2 Ostrowski Numeration System

We write the *continued fraction* of a real number x as a sequence of natural numbers $x = [a_0, a_1, a_2, a_3, \ldots]$ such that

$$x = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \ddots}}}$$

If the sequence is infinite and ultimately periodic with repeating part $a_i \ldots a_j$, then we write this as $x = [a_0, a_1, a_2, \ldots \overline{a_i, \ldots, a_j}]$.

We make use of the *convergents* of the continued fraction to define an Ostrowski-α numeration system, where α is some positive irrational number.

$$\frac{p_n}{q_n} = [a_0, a_1, a_2, \ldots, a_n]$$

The sequences p and q are given by the following recursive definition:

$$\begin{aligned} p_{-2} &= 0, \quad p_{-1} = 1, \quad p_n = a_n p_{n-1} + p_{n-2} \quad \text{for} \quad n > 0 \\ q_{-2} &= 1, \quad q_{-1} = 0, \quad q_n = a_n q_{n-1} + q_{n-2} \quad \text{for} \quad n > 0 \end{aligned} \tag{1}$$

The convergents approximate α, specifically:

$$\left| \alpha - \frac{p_n}{q_n} \right| < \frac{1}{q_n q_{n+1}} < \frac{1}{q_n^2} \quad \text{for all} \quad n.$$

Then every natural number n may be written as a sequence $b_j b_{j-1} \cdots b_0$ where each $b_i \in \mathbb{N}$, such that $b_0 < a_0$, $b_i \leq a_i$, and if $b_i = a_i$, then $b_{i-1} = 0$, and the following equation holds:

$$n = \sum_{i=0}^{j} b_j q_j$$

We call this sequence $b_j b_{j-1} \cdots b_0$ the *Ostrowski-α representation of n*. A word $b_j b_{j-1} \cdots b_0$ is a *valid* Ostrowski-α representation if $b_0 < a_0$, and for all $i > 0$, either $b_i < a_i$ or $b_i = a_i$ and $a_{i-1} = 0$. Each natural number has a unique valid Ostrowski-α representation [9].

For example, when $\alpha = \sqrt{2}$, the sequence (q_n) starts out $1, 2, 5, 12, \ldots$. We can write 14_{10} in the Ostrowski-$\sqrt{2}$ numeration system as: $0 \cdot 1 + 1 \cdot 2 + 0 \cdot 5 + 1 \cdot 12$, or $1010_{\sqrt{2}}$. Note there are two ways to write each number: least significant digit (LSD) and most significant digit MSD. Above, $1010_{\sqrt{2}} = 14_{10}$ is written in the standard MSD form, however, our automata use LSD because it is more natural to create such automata.

8.1.3 Sturmian Words

We define *Sturmian words* as follows.

Consider the standard coordinate plane, and some irrational number $\alpha \in (0, 1)$. Draw the line $y = \alpha x$. We can define the *Sturmian words* via a *cutting sequence* given by this line in the standard coordinate plane; that is, as a sequence of letter corresponding to the grid lines crossed by the line. We define the word cut_α as follows: $\text{cut}_\alpha[n] = 0$ if the n-th line crossed by $y = \alpha x$ is a vertical line, and $\text{cut}_\alpha[n] = 1$ if it is a horizontal line.

Then the *characteristic Sturmian word with slope α*, C_α is given by $C_\alpha = \text{cut}_{\alpha/(1-\alpha)}$ [1, Theorem 9.2.1]. Note that the first index of C_α is taken to be 1, not 0. Using this procedure, we can see that C_φ, shown in Figure 5 starts out $10110101\ldots$.

There is an intimate connection between the Ostrowski-α numeration system and the characteristic Sturmian word with slope α, given by the following theorem ([1, Theorem 9.1.15]).

Theorem 8.1. *Let $n \geq 1$ be an integer with Ostrowski-α representation: $b_j b_{j-1} \cdots b_0$. Then $C_\alpha[n] = 1$ if and only if $b_j b_{j-1} \cdots b_0$ ends in an odd number of 0's.*

In particular, this means that C_α is an automatic sequence, so we can create an automaton $\mathcal{C}_\alpha$ such that $\mathcal{C}(n)$ is true if and only if the n-th digit of C_α is 1. We define:

$$\mathcal{C}_\alpha[n] = \begin{cases} 1 & \text{if } C_\alpha(n) \\ 0 & \text{otherwise} \end{cases}$$

8.1.4 Conventions

Note that all quantifiers over Latin letters (e.g., i, j, k, ℓ, n, m) are over the natural numbers, and quantifiers over the Greek letters (e.g., α) are over real numbers in $(0, 1)$ which produce Ostrowski numeration systems: in particular, they must be irrational and their continued fraction cannot start with 1.

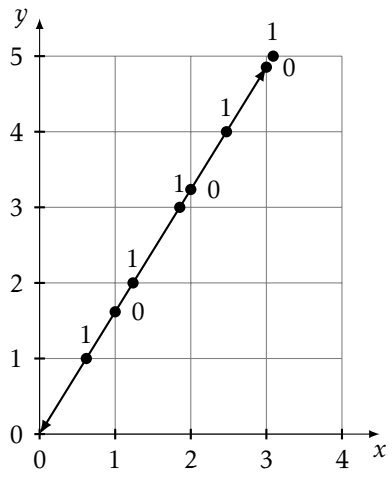

Figure 5: The Sturmian word C_φ, defined by the cutting sequence cut_φ with $\varphi = \frac{1+\sqrt{5}}{2}$

8.2 Representing Ostrowski Numeration Sytems

In order to implement a decision procedure for Sturmian words, we need automata to recognize the following properties of Ostrowski-α numeration systems:

- `bco_valid(a,x)`: If x is a valid representation in the Ostrowski-α numeration system, given α. We also define `ostrowski(x,a) :=bco_valid(a,x)` for convenience.
- `bco_leq(x,y)`: For two valid representations x and y, if $x \leq y$.
- `bco_adder(a,x,y,z)`: For three valid representations, x, y, and z, if $x + y = z$, given α.

We encode representations of natural numbers in LSD in Ostrowski numeration systems using $\Sigma = \{0, 1, 2\}$ with each digit as a binary number in LSD and digits separated by 2. Note that the second property can be decided without the parameter α, but the first and third require it.

It is easy to encode the first two properties, but encoding a general addition automaton for Ostrowski numeration systems is more difficult.

8.2.1 Derived Predicates

Using the basic definitions above, we can encode other predicates that are quite useful:

- `bco_zero(x)`: Checking whether x is the representation of 0, defined as z such that $\forall y.z \leq y$.
- `bco_eq(x,y)`: Checking whether $x = y$, defined as $x \leq y \wedge y \leq x$.
- `bco_succ(a,x,y)`: The *successor* of x (i.e., $x + 1$), defined as the number y such that $x < y$ and $\forall z.z \leq x \vee y \leq z$.

8.2.2 Proof of Correctness

Using Pecan, we can prove that the automata we have created are correct in a number of ways, by checking basic properties of addition and the natural numbers.

Because we have defined the successor of x without reference to the addition automaton, we can use it to check the correctness of addition. We check that our addition automaton conforms to the standard definition of addition on the natural numbers:

$$0 + y = y$$
$$s(x) + y = s(x + y)$$

where $s(a)$ denotes the successor of a. We check that our adder satisfies this property using the following Pecan code:

```
Restrict x,y,z are ostrowski(a).
Theorem ("Addition base case.", {
    ∀a. ∀x,y,z.
    if bco_zero(x) then
        (bco_adder(a, x, y, z) <=> bco_eq(y, z))
}).

Theorem ("Addition inductive case.", {
    ∀a. ∀x,y,z,u,v.
    if (bco_succ(a,u,x) & bco_succ(a,v,z)) then
        (bco_adder(a,x,y,z) <=> bco_adder(a,u,y,v))
}).
```

We also verify that every natural number has a unique successor in each Ostrowski-α numeration system, and every natural number (apart from 0) have a predecessor, where the predecessor of y is x such that `bco_succ(a,x,y)` holds. We also prove some properties about the relationship between `bco_leq` and `bco_adder`.

```
Theorem ("Every valid Ostrowski-a representation has a successor.", {
    ∀ a, x.
    if bco_valid(a, x) then
        ∃ u. bco_succ(a,x,u)
}).

Theorem ("All natural numbers other than 0 have a predecessor.", {
    ∀ a, x.
    if bco_valid(a,x) then
        (bco_zero(x) | (∃ u. bco_succ(a,u,x)))
}).

Theorem ("For all x, y, and z, we have x <= y iff x + z <= y + z", {
    ∀ a. ∀ x,y,z,u,v.
    if (bco_valid3(a,x,y,z) & bco_valid2(a,u,v) & bco_adder(a,x,z,u) & bco_adder(a,y,z,
        (bco_leq(x, y) <=> bco_leq(u, v))
}).

Theorem ("If x <= y, then there is some z so that x + z = y.", {
    ∀ a. ∀ x,y.
    if bco_valid2(a,x,y) then
        (bco_leq(x,y) <=> ∃ z. bco_adder(a,x,z,y))
}).
```

There are many other simple correctness properties we check, such as $\leq$ being a total order, which can be found in our project repository at https://github.com/ReedOei/SturmianWords.

8.3 Results

As the automata tend to be large, almost all having roughly 100 or more states and hundreds of edges, displaying them graphically provides little insight, and so we omit such diagrams below. Note that the exact predicates and many of the intermediate automata are available in the following GitHub repository: https://github.com/ReedOei/SturmianWords.

8.3.1 Periodicity

We begin with a well-known result about Sturmian words: that they are not eventually periodic. In fact, a binary ω-word is a Sturmian word if and only if it is a balanced, aperiodic binary word ([7, Theorem 2.1.5]).

Definition 8.2. An ω-word w is *eventually periodic with period p* for some $p > 0$ if there is some natural number n so that $w[i] = w[i+p]$ for all $i > n$.

Theorem 8.3. *No Sturmian word is eventually periodic.*

Proof. Using Pecan, we construct an automaton recognizing the following property:

$$\texttt{eventually\_periodic}(\alpha, p) := p > 0 \wedge \exists n. \forall i > n. C_\alpha[i] = C_\alpha[i+p]$$

In Pecan, this can be written as:

```
Restrict a is bco_standard.
Restrict i, n, p are ostrowski(a).
eventually_periodic(a, p) :=
    p > 0 & ∃n. ∀i. if i > n then $C[i] = $C[i + p]
```

Finally, we use Pecan to construct the automaton recognizing the following:

$$\forall \alpha, p > 0. \neg \texttt{eventually\_periodic}(\alpha, p)$$

In Pecan, this is:

```
Theorem ("Sturmian words are not eventually periodic", {
    ∀ a, p. if p > 0 then !eventually_periodic(a, p)
}) .
```

The resulting automaton is the $\top$-automaton, which proves the theorem. □

In the rest of this paper, we omit the exact input given to Pecan, and instead write the property definitions in a more typical style for readability. Please refer to the repository linked above for the exact inputs.

8.3.2 Powers

Next, we prove the follow results about *powers* of Sturmian words.

Definition 8.4. A finite subword x of a (finite or ω) word w is a *n-th power* if $x = y^n$ for some finite word y. We call a 2nd power a *square*, and a 3rd power a *cube*.

A word that does not contain n-th powers is called *n-th power free* (or *square-free*, *cube-free* when appropriate).

Theorem 8.5. *Sturmian words contain squares.*

Proof. Using Pecan, we construct an automaton recognizing the following property, stating that there is a square of length n starting at $C_\alpha[i]$.

$$\texttt{square}(\alpha, i, n) := n > 0 \wedge i > 0 \wedge \forall j. i \leq j < i + n. C_\alpha[j] = C_\alpha[j + n]$$

The resulting automaton has 80 states and 400 edges. Finally, we use Pecan to construct the automaton recognizing the following:

$$\forall \alpha. \exists i, n. \texttt{square}(\alpha, i, n)$$

This automaton is the $\top$-automaton, which proves the theorem. □

In fact, all sufficiently long (i.e., longer than 4 letters) binary words contain squares. A binary word may start with 0 or 1, and without loss of generality, suppose it starts with 0. Then the next letter may not be 0, or it would contain a square; similarly, the next letter must be 0, and then the final letter will make the word contain a square, regardless of what we choose.

However, it is still useful to have create an automaton for recognizing squares, because we use it to build automata recognizing higher-powers.

Sturmian words have an interesting property related to squares, namely that they start with arbitrarily long squares ([3, Theorem 1]).

Theorem 8.6. *Sturmian words start with arbitrarily long squares.*

Proof. Using Pecan and the automaton for squares that we constructed earlier, we define

$$\forall \alpha. \forall n. \exists m > n. \texttt{square}(a, 1, m)$$

This automaton is the $\top$-automaton, which proves the theorem. □

The following two theorems are consequences of [2, Theorem 4], which we can prove using Pecan.

Theorem 8.7. *All Sturmian words contain cubes.*

Proof. We construct an automaton recognizing the following property, stating that there is a cube of length n starting at $C_\alpha[i]$.

We can use the previously defined square to define cubes, as follows:

$$\texttt{cube}(\alpha, i, n) := \texttt{square}(\alpha, i, n) \wedge \texttt{square}(\alpha, i + n, n)$$

Finally, we use Pecan to construct the automaton recognizing the following:

$$\forall \alpha. \exists i, n. \texttt{square}(\alpha, i, n)$$

This automaton is the $\top$-automaton, which proves the theorem. □

Similar to squares, we have the following property about cubic prefixes

Theorem 8.8. *A Sturmian word starts with arbitrarily long cubes if and only if its continued fraction is not eventually 1.*

Proof. First, we manually build an automaton recognizing α such that the continued fraction of α is not eventually one, called `eventually_one`.

Then we use Pecan to construct the automaton recognizing the following:

$$\forall \alpha. (\neg \texttt{eventually\_one}(\alpha)) \iff \forall n. \exists m > n. \texttt{cube}(a, 1, m)$$

This automaton is the $\top$-automaton, which proves the theorem. □

Theorem 8.9. *All Sturmian words whose continued fraction is not eventually* 1 *contain fourth powers.*

Proof. We construct an automaton recognizing the following property, stating that there is a fourth power of length n starting at $C_\alpha[i]$. We can use the previously defined square and cube definitions to define fourth powers, as follows:

$$\texttt{fourth\_pow}(\alpha, i, n) := \texttt{square}(\alpha, i, n) \wedge \texttt{cube}(\alpha, i+n, n)$$

Finally, we use Pecan to construct the automaton recognizing the following:

$$\forall \alpha. \neg \texttt{eventually\_one}(\alpha) \implies \exists i, n. \texttt{square}(\alpha, i, n)$$

This automaton is the $\top$-automaton, which proves the theorem. □

The converse is not true, and using Pecan, we can easily see why not. We can ask Pecan for counterexamples to the converse using the following commands.

```
Restrict i, n are ostrowski(a).
has_fourth_pow(a) := ∃ i, n. n > 0 & fourth_pow(a, i, n)
Example (ostrowskiFormat, {
    bco_standard(a) & eventually_one(a) & has_fourth_pow(a)
}).
```

Pecan responds with:

```
[(a,[6][3]([1])^ω)]
```

Meaning that $\alpha = [6, 3, \overline{1}]$ is a counterexample; indeed, C_α starts as $000001\ldots$, so there is a fourth power immediately at the beginning of it. Using a sufficiently high starting coefficient, $\alpha = [a_0, a_1, \ldots]$, we can get any power, because the representations for x with $0 < x < a_0$ will all start with an even number of zeros, and so $C_\alpha[x] = 0$.

We now move on to another interesting sort of "repetition", called an antisquare.

Definition 8.10. A finite subword x of a (finite or ω) word w is an antisquare if $x = y\bar{y}$ for some finite word y.

Theorem 8.11. *All Sturmian words contain finitely many antisquares.*

Proof. Similar to squares, we construct an automaton recognizing the following property:

$$\texttt{antisquare}(\alpha, i, n) := n > 0 \wedge i > 0 \wedge \forall j. i \le j < i + n \implies C_\alpha[j] \neq C_\alpha[j+n]$$

Finally, we use Pecan to construct the automaton recognizing the following:

$$\forall \alpha. \exists m. \forall n, i. \texttt{antisquare}(\alpha, i, n) \implies n < m$$

This automaton is the $\top$-automaton, which proves the theorem. □

8.3.3 Palindromes

Theorem 8.12. *All Sturmian words start with arbitrarily long palindromes.*

Proof. We construct an automaton recognizing the following property:

$$\texttt{palindrome\_prefix}(\alpha, n) := n > 0 \wedge \forall i. 0 < i \le n \implies C_\alpha[i] = C_\alpha[n - i + 1]$$

The resulting automaton has 95 states and 383 edges.

Finally, we use Pecan to construct the automaton recognizing the following:

$$\forall \alpha. \forall m. \exists n. n > m \wedge \texttt{palindrome\_prefix}(a, n)$$

This automaton is the $\top$-automaton, which proves the theorem.

□

References

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

[2] David Damanik and Daniel Lenz. "Powers in Sturmian sequences". In: *European Journal of Combinatorics* 24.4 (2003), pp. 377–390. ISSN: 0195-6698. DOI: https://doi.org/10.1016/S0195-6698(03)00026-X. URL: http://www.sciencedirect.com/science/article/pii/S019566980300026X.

[3] Dubickas, Arturas. "Squares and cubes in Sturmian sequences". In: *RAIRO-Theor. Inf. Appl.* 43.3 (2009), pp. 615–624. DOI: 10.1051/ita/2009005. URL: https://doi.org/10.1051/ita/2009005.

[4] Alexandre Duret-Lutz et al. "Spot 2.0 — a framework for LTL and ω-automata manipulation". In: *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*. Vol. 9938. Lecture Notes in Computer Science. Springer, Oct. 2016, pp. 122–129. DOI: 10.1007/978-3-319-46520-3_8.

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

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

[7] M. Lothaire. *Algebraic combinatorics on words*. English. Vol. 90. Cambridge: Cambridge University Press, 2002, pp. xiii + 504. ISBN: 0-521-81220-8/hbk.

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

[9] Alexander Ostrowski. "Bemerkungen zur Theorie der Diophantischen Approximationen". In: *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg* 1.1 (Dec. 1922), pp. 77–98. ISSN: 1865-8784. DOI: 10.1007/BF02940581. URL: https://doi.org/10.1007/BF02940581.

[10] 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.