[PROVE]

24th January 2021

1 Introduction

This document is work in progress. All source code as well as the most recent version of this document can be found on GitHub (see: `https://github.com/g-regex/prove/blob/main/doc/doc.pdf`). The purpose of this document to introduce notation, vocabulary and some underlying principles in order to facilitate efficient communication about this project. Furthermore this document could serve as an introduction to the project, if anybody wants to join.

2 Terminology

Let us start with some basic definitions

- A *statement* is a concatenation of one or more pair(s) of square brackets enclosing another statement, a formula or an identifier.
- A *formula* is a concatenation of statements and formulators. The empty string is a special case of a formula. Apart from this special case every formula must contain at least one statement and at least one formulator.
- A *formulator* is a sequence of legal (see later definition) characters. A formulator does not contain any square brackets but is delimited by a closing square bracket or the beginning of the file at its beginning and an opening square bracket or the end of the file at its end.
- An *identifier* is a sequence of legal characters and is delimited by an opening square bracket at its beginning and by a closing square bracket at its end.
- To refer to formulators or identifiers more generally, we will use the term *symbol* in this document.
- To refer to statements or formulas more generally, we will use the term *expression* in this document.
- To refer to a formula, which is not an equality and not an implication, the term *ordinary formula* will be used throughout the document.
- When we speak of a *character* in this document, we mean a uppercase or lowercase Latin letter, a number or a special character from a list that may be extended. Opening and closing square brackets, the literals used in the reserved formulators `=`, `=>` and the negation `!` are not referred to as characters in this document.
- In the current implementation and negated statement is still a statement and not a formula. The `!` is currently not a formulator - probably this has to be discussed.
- The terms *variable* and *constant* can be used to refer to an identifier. Which term is to be used depends on the context, we are using it in (see later definition).
- The term *real constant* refers to a formula, comprised of exactly one formulator and exactly one empty statement.

3 Syntax

3.1 EBNF

The syntax of the [PROVE] language can conveniently be described by means of the extended Backus-Naur form(EBNF), which is a meta syntax notation defined in the ISO/IEC 14977 to recursively and unambiguously describe nested patterns by grouping syntactic elements to so called ⟨production⟩s. It must always be possible to 'unpack' a production in a unique and finite way, resulting in a (sequence of) *terminal* symbols, indicated by quotation marks. In order to correctly interpret the [PROVE]-EBNF it is sufficient to understand the meaning of the following meta characters:

|	indicates 'or'
(...)	groups productions and/or terminals together; only useful in combination with |
[...]	indicates optionality
{...}	indicates zero or more repetitions

Using this notation the [PROVE] language can be described by the following EBNF:

$$
\begin{aligned}
\langle \texttt{expr} \rangle &= \langle \texttt{statement} \rangle \Big| \langle \texttt{formula} \rangle \\
\langle \texttt{formula} \rangle &= \Big\{ \langle \texttt{statement} \rangle \langle \texttt{formulator} \rangle \Big| \langle \texttt{formulator} \rangle \langle \texttt{statement} \rangle \Big\} \\
\langle \texttt{statement} \rangle &= \Big[\texttt{"!"} \Big] \ \texttt{"["} \ \Big(\langle \texttt{expr} \rangle \Big| \langle \texttt{symbol} \rangle \Big) \texttt{"]"} \Big\{ \Big[\texttt{"!"} \Big] \ \texttt{"["} \ \Big(\langle \texttt{expr} \rangle \Big| \langle \texttt{symbol} \rangle \Big) \ \texttt{"]"} \ \Big\} \\
\langle \texttt{formulator} \rangle &= \texttt{"="} \Big| \texttt{"=>"} \Big| \langle \texttt{symbol} \rangle \\
\langle \texttt{symbol} \rangle &= \langle \texttt{character} \rangle \Big\{ \langle \texttt{character} \rangle \Big\} \\
\langle \texttt{character} \rangle &= \Big(\texttt{"a"} \Big| \texttt{"b"} \Big| \ldots \Big| \texttt{"z"} \Big| \texttt{"A"} \Big| \texttt{"B"} \Big| \ldots \Big| \texttt{"Z"} \Big| \texttt{"0"} \Big| \texttt{"1"} \Big| \ldots \Big| \texttt{"9"} \Big| \texttt{"+"} \Big| \texttt{"-"} \Big| \texttt{"/"} \Big| \texttt{"*"} \Big| \texttt{"\%"} \Big| \texttt{"\^{}"} \Big| \texttt{"\&"} \Big| \texttt{"."} \Big| \texttt{"?"} \Big| \texttt{":"} \Big)
\end{aligned}
$$

Every EBNF needs to have an initial production as a starting point. It is convention that this distinguished production is to be listed first, so in this case, ⟨expr⟩ is the initial production. As one might notice a valid [PROVE] file might contain either a statement or a formula. This design choice has been made in order to account for axioms and theorems that might be given in the human readable form of the [PROVE] language. Proofs on the other hand will be given in the form of a formula, having only `=>` formulators at its outermost level. We could easily think of other formulas, which will not satisfy this requirement and thereby neither constitute an axiom/theorem nor a proof. A [PROVE] file containing such a formula would be syntactically correct but would violate semantic rules, which will be established later.

3.1.1 Examples

Here are some examples of productions and some motivations for the design of the syntax. We will work our way up from bottom to top through the EBNF.

- A ⟨`character`⟩ can be any letter, number or one of the 10 special characters as indicated in the EBNF. Examples of valid ⟨`character`⟩s are `a`, `T`, `7`, `&`.
- A ⟨`symbol`⟩ is a sequence of one or more characters. Examples of valid ⟨`symbols`⟩s are: `a7we`, `T`, `76`, `&q`. Note that `T` is a ⟨`symbol`⟩ and a ⟨`character`⟩ at the same time. This does not mean that the grammar is ambiguous. A single `T` will always be interpreted in this way, but it depends on the context, whether we are interested in the fact of `T` being a ⟨`symbol`⟩ or `T` being a ⟨`character`⟩.
- A ⟨`formulator`⟩ is either a ⟨`symbol`⟩ or any one of the terminals (fixed, not "unpack-able" elements of an EBNF) `=` or `=>`. Examples of valid ⟨`formulator`⟩s are `a7we`, `T`, `=`, `=>`. Note that `=T` is not a valid ⟨`formulator`⟩. Seeing that `T` can be a ⟨`formulator`⟩ (comprised of a ⟨`symbol`⟩) in one case and just a ⟨`symbol`⟩ (not packed inside a ⟨`formulator`⟩) in another case, one might worry about ambiguity. However, having a close look at the EBNF, we see that ⟨`formulator`⟩s only occur in between ⟨`statement`⟩s and ⟨`string`⟩s only within ⟨`statement`⟩s. The user of the language has great freedom in naming formulators and identifiers (TODO: define formulators and identifiers in this document, i.e. not the productions, but what they represent), so it is up to the user to choose them wisely. It could for example be confusing to name a formulator `7`, but `add7` makes intuitively sense.
- A ⟨`statement`⟩ is a concatenation of one or more pairs of square brackets each containing either a ⟨`formula`⟩, a ⟨`symbol`⟩ or a ⟨`statement`⟩. Examples of valid ⟨`statement`⟩s are `[]`, `[a][b][[a]in[b]]`, `[[]]`, `[[a]or[b]]`.
- A ⟨`formula`⟩ is comprised of at least one ⟨`statement`⟩ and at least one ⟨`formulator`⟩. The order of occurrence does not matter. Examples of valid ⟨`formula`⟩e are `[a]in[b]`, `[a]=[b][c]`, `[[a]a7we[b]]=>[c]`.

4 Semantic

When generating a couple of syntactically correct snippets of `[PROVE]` code, one realises that not all code adhering to the rules of the EBNF makes sense - at least not without introducing more conventions.

Consider the following code:

```
[a]and[b]=[c]=>[b][c]
```

The use of `=` and `=>` is ambiguous in this context, since `=` could refer to everything on its right hand side or just to `[c]`. Similarly `=>` could refer to everything to its left hand side or just to `[c]`. This problem could be addressed by introducing an order of precedence (e.g. non reserved formulators, `=`, `=>`). However, the implications of this approach should be carefully considered. Precedence can easily be encoded in the EBNF, without changing the set of valid code patterns. By doing so we could elegantly encode a part of the language's meaning in its syntactic structure. As tempting as this idea might be, I can see a negative side effect coming along with this approach. Introducing precedence between formulators will set brackets *implicitly.* Since the visibility and meaning of identifiers (variables/constants) depends on the depth of nesting of statements (i.e. on brackets), the approach of precedence might lead to counter-intuitive code. Therefore I recommend against the precedence approach.

Instead I suggest the following semantic rule, which will address this issue:

- `=`, `=>` and other formulators must not be mixed

Consider the following code:

```
[a][b]=>[c]
```

Translated to English this means 'Let a and b be identifiers, then we have an identifier c'. We do not know, where c comes from, since it was not introduced before. Therefore I suggest the following rule:

- New identifiers must only be introduced in a part of a formula, which is reserved for assumptions.

To determine, which statements within a formula are assumptions and which statements are conclusions, I'd like to put the following three rules up for discussion:

- If the formula is an implication (i.e., all formulators are `=>`), then every statement occurring before the first formulator is an assumption.
- Every statement in an equality (i.e., all formulators are `=`) is an assumption, from which all other statements can be concluded. (Remark: An equality has all the properties a two-sided implication would have, but in addition to that in also allows substituting one statement with another)
- If a statement is an assumption, all statements contained in it are assumptions as well.

It might also be convenient to restrict the form an equality can have. Especially with respect to a straight forward implementation, it would make sense to only let single statements (i.e. only one pair of square brackets at the outermost level) to be equal to each other. Therefore I suggest the following rules:

- Equalities are only such formulas, which are a concatenation of single statements and `=`, where no single statement is next to another single statement and no `=` is next to another `=`
- An equality must start and end with a single statement.

Further I suggest the following rule for implications for obvious reasons:

- `=>` must not appear at the end of a ⟨`statement`⟩

5 Representation

Let us have a look at the following statement:

Source-Code 1: Example 1

```
1 [a][b][[a]rel[b]]
```

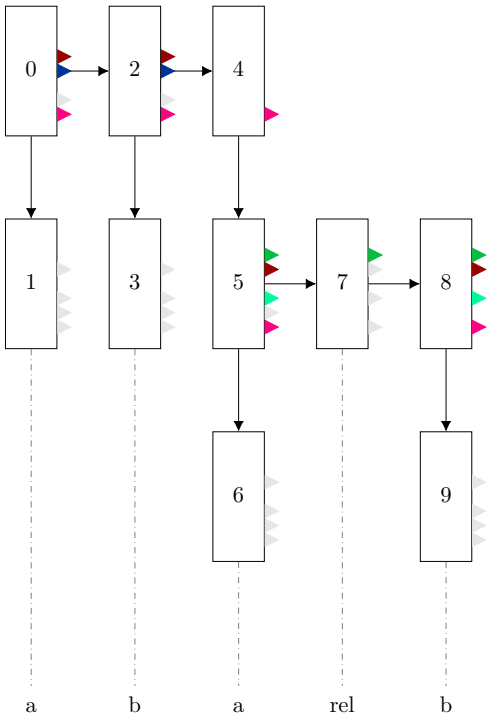
An intuitive approach for representing the structure of this proof is by means of a graph (in particular a special planar tree).

Figure 1: Representation of Source-Code 1

(a) Legend

- `IMPL`: node is part of an implication
- `EQTY`: node is part of an equality
- `FMLA`: node is part of an ordinary formula
- `ASMP`: node is an assumption
- `NEWC`: node contains a newly introduced constant
- `LOCK`: `ASMP` flag is locked for the entire subtree
- `FRST`: node is positioned before first occurrence of an implication formulator in that formula
- `TRUE`: statement is true
- subtree contains at least one variable
- property is not applicable to that node

(b) Graph of Source-Code 1



6 Verification

6.1 Variables and constants

TODO

6.2 Negation

TODO

7 Output

TODO